SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2013; **23**:315–350
Published online 19 March 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1471

Parallel mutation testing

Pedro Reales Mateo*,† and Macario Polo Usaola

Universidad de Castilla-La Mancha, Ciudad Real, Spain

SUMMARY

Despite the existing techniques to reduce the costs of mutation analysis, the computational cost to apply mutation testing with large applications can be very high. One effective technique to improve the efficiency of mutation without losing effectiveness is parallel execution, where mutants and tests are executed in parallel processors, reducing the total time needed to perform mutation analysis. This paper presents a study of this technique adapted to current technologies. Five algorithms to execute mutants in parallel are analysed with three studies that use different network configurations and different number of processors with diverse characteristics. The experiments are performed with Bacterio^P, a tool that is also presented. Unlike previous studies about parallel mutant execution, which date from the mid-1990s, in the studies in this paper, the communication time in parallel systems no longer acts as a bottleneck. Thus, dynamic strategies, which require more communication, combined with other mutant cost reduction techniques, are the best strategies to run mutants in parallel. Copyright © 2012 John Wiley & Sons, Ltd.

Received 1 June 2011; Revised 24 February 2012; Accepted 28 February 2012

KEY WORDS: mutation testing; parallel execution

1. INTRODUCTION

Program mutation is an effective testing technique to test systems intensively. In mutation testing, mutants of the program under test are generated by the application of one or more mutation operators. Each mutant has one or more code perturbations that, sometimes, can be interpreted as faults. The number of faults found on the system (i.e. the number of mutants killed) is a good adequacy criterion for the test suite. Although these changes are simple, the coupling effect [1] indicates that a test suite sensitive enough to find them will be able to find also the complex errors in the original system. Compared with other white box testing techniques, mutation testing means that the tester must design tests to kill mutants, instead of tests that execute some parts of the code.

A mutation analysis involves three main tasks. The first task is to create mutants, which are automatically created through mutation operators (well-defined rules to make syntactic changes in the code). The second task is to execute the program under test and its mutants against the test cases. The goal of this task is to determine how many mutants are killed by the tests (a mutant is considered killed when the output of the original system is different from the output of the mutant). The third task is to analyse the results. In this task, the mutation score (Equation 1) is calculated to measure the mutation adequacy of the test suite.

When mutation is applied to large systems, a huge amount of mutants can be generated. Despite existing cost reduction techniques [2], the problem of the required computational time to execute tests against the original system and their mutants still remains. The costs of mutation testing mainly

*Correspondence to: Pedro Reales Mateo, Tecnologías y Sistemas de Información, Universidad de Castilla-La Mancha, Ciudad Real, Spain.

†E-mail: pedro.reales@uclm.es

$MS(P,TS) = K / (T - EQ)$	...where:	P = program under test TS = test suite K = number of killed mutants T = total number of mutants EQ = number of equivalent mutants

Equation 1. Mutation score.

depend on the number of mutants generated and on the number of test cases. In a single sequential process, the total computational time of mutation testing is given by Equation 2.

In the equation, *Gt*Nm* represents the time spent in mutant generation, whereas *Et*Ntc*(Nm+1)* is the time devoted to execute the tests against all the mutants and the original system (which must be executed at least once). The execution time (*Et*) is always much higher than the generation time (*Gt*), and this is the reason that researchers have addressed their research efforts to reduce *Et, Nm* or *Ntc*:

- Regarding *Et*, mutant schemata [3] makes it possible to speed up the execution thanks to the inclusion of all the mutants in a single file. This avoids the continuous load of the mutant file on the memory and avoids the launching of a new process to execute each program version.
- Regarding *Nm*, techniques such as the random selection of mutants [4], selective mutation or constrained mutation [5–8] and higher-order mutation [9] produce fewer mutants, which has a positive influence on the total execution time.
- Regarding *Ntc*, the research focuses on reducing the test suite size: given a test suite *TS*, the goal is to obtain a test suite *TS'* that obtains the same coverage (in this case, the same mutation score) as *TS* such that $|TS'| \leq |TS|$.
- Regarding *Gt*, techniques like byte-code translation [10], which remove the compilation tasks, can reduce the total time of the process.

Besides the described techniques, parallel execution tries to reduce the overall time Mt by distributing the executions across different processors. In this context, $Mt = Gt*Nm + \max\{Et_i*Ntc_i*(Nm_i + 1)\}$.

For instance, let us suppose a large application that will be tested with mutation. Let be $Nm = 1\ 000\ 000$ (number of mutants), $Gt = 0.001$ seconds (generation time of one mutant), $Ntc = 1000$ (number of test cases), $Et = 0.001$ s (execution time of one test case). The total time is $0.001*1\ 000\ 000 + 0.001*1000*1\ 000\ 001 = 1\ 001\ 001$ s $= 278.05$ h $= 11.5$ days in the worst case, which is a prohibitive time just to perform a mutation analysis. However, if the mutation analysis runs on a cluster with 200 parallel processors, the execution time can be reduced from 11.5 days to 4 or 5 h, depending on the distribution algorithm and on the nature of the test. Thus, to study parallel mutation testing techniques is a necessary task to reduce mutant execution times without losing effectiveness.

The expression in Equation 2 represents the execution time in the worst case scenario; that is, when all the test cases are executed against all the mutants. In practice, mutant execution is performed iteratively, and in each iteration, mutants that are killed are removed from the list of mutants, which makes it possible to execute the following test cases against only the mutants remaining alive [11]. Although this technique may obtain important time savings, the number of mutants is still the most influential factor in the execution time.

$Mt=Gt*Nm+ Et*Ntc*(Nm+1)$	...where:	Mt=mutation time Gt=generation time of one mutant Et=execution time of one test case Ntc=number of test cases Nm=number of mutants

Equation 2. Computational cost of mutation testing.

Some authors estimate that the number of mutants generated is in the order of the square of the number of variable references in the program [12], although this quantity depends on the mutation operators applied to the program under test. Figure 1 shows how the number of mutants grows more than linearly with the number of lines of code of four packages with different sizes of the Apache math project [13]. For the test of very large systems (millions of lines of code), the 'mutation tester' should focus on the most critical parts of the system and apply some additional cost reduction technique, such as selective mutation [6] or mutant sampling [14], to reduce the number of mutants (it has been shown that a mutation score of 100% reached on 10% of mutants is nearly adequate for a full mutation analysis [4]). Moreover, depending on the features to test, the tester should select the appropriate mutation operators. For instance, to test functional features, the tester should select traditional mutations operators (as the reduced set specified by Offutt *et al.* [6]), or to test integration features, the tester should select integration mutation operators (as described by Ma *et al.* [15]).

The size and complexity of the system under test also influence the execution costs due to the number of test cases; the larger or more complex the application, the more test cases are required to achieve adequate coverage. Figure 2 compares the generation and execution times in seconds of the four applications used in the experimental section of this article. Whereas the mutant generation time grows slowly, the execution time grows exponentially with the number of mutants and test cases, which in turn depends on the application size.

Because the execution time is the most significant of the two, this article only focuses on the execution phase. The paper presents a study of the mutant parallel execution technique, which is suitable to reduce the computational cost of the execution phase. In this study, five distribution algorithms were analysed in several environments to determine the best strategy to parallelize the execution phase. Also, two other studies were performed to analyse the effect of homogenous and heterogeneous environments and the influence of the environment size on the speedup.

The remainder of this paper is organized as follows. Section 2 describes some work on parallel mutation testing and demonstrates the need for new studies of this technique. Section 3 describes the distribution algorithms used in the studies. The tool used to perform the experiments is presented in Section 4. Section 5 shows the design of the experiments. Section 6 describes the obtained results, and finally, Section 7 presents some conclusions and future work.

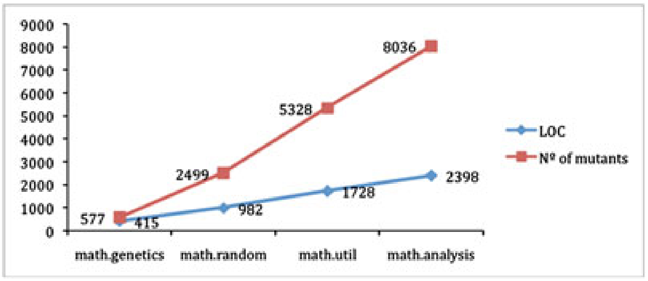


Figure 1. Relation between lines of code and number of mutants.

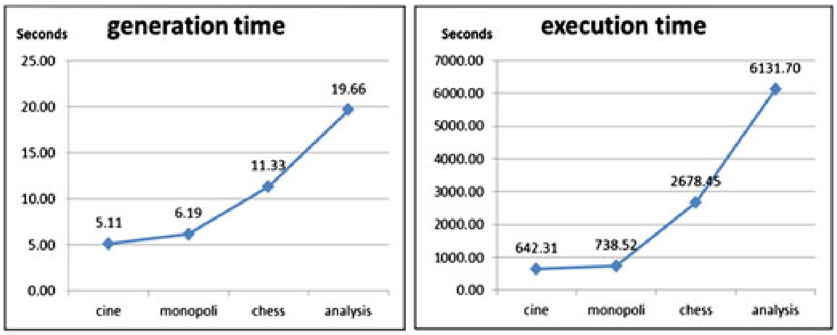


Figure 2. Generation and execution times (in seconds).

2. PARALLEL MUTATION TESTING

An important cost reduction technique is parallel execution [16, 17], which is widely used in other contexts, such as natural process simulations (e.g. weather forecasting and chemical reaction simulation). There are several works about the parallel execution of mutants in literature. In the survey about mutation by Jia and Harman [18], the authors identified three lines of investigation: execution of mutants (i) in single instruction, multiple data machines (SIMD), (ii) in multiple instruction, multiple data (MIMD) machines and (iii) with optimized serial algorithms.

The earliest work was published in 1988 by Mathur and Krauser [19]. In this study, the authors proposed to execute several mutants in a vector processor. This work is only applicable with mutations that replace variables. Specifically, the algorithms described in it were designed for mutants obtained with the scalar variable replacement (SVR) mutation operator (Offutt *et al.* [6] demonstrated that the SVR operator generates the largest number of mutants). The authors' basic idea was to map simultaneous executions of SVR mutants to a sequence of vector instructions, which can be executed in parallel in a vector processor. Although this work improves the execution efficiency, the SVR mutation operator is not widely used today because some empirical studies [6, 7] determined the minimal and sufficient set of mutation operators, and the SVR mutation operator was not included. This work has thus lost value over time.

Together with Rego [20], the same authors extended their work with the program unification technique [21] to apply the technique to mutants proceeding from any operator. To validate and evaluate their proposal, they made simulations with the models developed. In their studies, they indicated that the major bottleneck observed was the compilation, which is not a problem today thanks to mutation techniques such as mutant schemata [3] and byte-code translation [10].

The second line of research was based on MIMD machines. The first study about parallel mutation with MIMD machines is the work by Choi and Mathur [17]. These authors presented the P^{m}othra tool that is an adaptation of the Mothra tool [22] for the Ncube/7 Hypercube machine. The P^{m}othra tool has an architecture based on modules that are executed in the nodes of the hypercube. It is based on a mutant generator, a mutant compiler, a mutant scheduler and several test case servers and mutant executors. This tool uses a dynamic distribution algorithm that executes a mutant when a node of the hypercube becomes available. The tool also has communications between the mutant executor nodes and the test case servers, because the executor nodes order test cases and expected outputs to the test case servers.

In the study, Choi and Mathur looked at the effect of the number of servers in the hypercube and determined that a small number of servers results in mutant executors waiting during communications with the test case server, but a high number of servers results in many nodes of the hypercube not being available and an increase in the total time. In this study, the communication time has a high impact on the efficiency of the technique.

Another important work, proposed by Offutt *et al.* [16], presented the HyperMothra tool, an adaptation of the Mothra tool [22] to be executed in the Intel iPSC/2 hypercube machine with 16 processors. The HyperMothra tool was designed to generate mutants for Fortran systems with 22 mutation operators and interpret them in parallel in the hypercube.

In the study, the authors looked at three different static distribution algorithms that split the set of mutants: (i) distributing mutants in the original order; (ii) distributing mutants randomly and uniformly; and (iii) uniform distribution of mutants by mutation type. They also used different configurations of the hypercube to have different sizes in the number of nodes. The authors concluded that this approach only achieved a high speedup with large applications, meaning that the parallel execution technique only increases the efficiency with large applications. With respect to static distribution algorithms, the authors determined that the most efficient algorithm distributes mutants by mutation type because it achieves the highest speedup. An important conclusion of this work was that communications between host and nodes formed the bottleneck of the technique and had a significant influence on the speedup (a similar conclusion to that of Choi and Mathur [17]). This means that other dynamic distribution algorithms (such as demand-driven strategies) that require more communication would achieve a lower speedup. In this article, besides mutant distribution strategies, strategies for test case distribution with both dynamic and static algorithms are analysed too.

The last line of research in parallel mutation testing is related to optimized serial algorithms. Fleyshgakker and Weiss [23, 24] proposed some algorithms to reduce the number of executions and improve the efficiency of the mutation process. The algorithms were not implemented for parallel execution. However, according to the authors, their structure makes it easily parallelizable.

To the best of our knowledge and according to the survey by Jia and Harman [18], the most recent work related to 'parallel mutation' is from 1994 [24]. Since that year, programming languages, networks and processors have evolved a great deal: in 1994, for example, the first Pentium processor appeared with a speed of 200 MHz. Currently on the market, there are processors like *Intel Core*$^{\text{TM}}$ *2 Extreme quad-core*, with four cores and 5 GHz of speed. Regarding software, new programming languages such as Java and those in .NET currently exist, whose characteristics of semi-interpretation allow engineers to take advantage of more new mutation techniques, such as byte-code translation [10]. In the works described in this section, all the mutation studies used programs written in Fortran.

Additionally, as several authors have pointed out [2, 25], mutation can move beyond being a research testing technique and is ready to be accepted by the industry. This fact, together with current technologies, networks, computing clouds and so on, leads us to think that this is an excellent moment to analyse the possibilities of parallel computation for mutation testing.

3. WORKLOAD DISTRIBUTION ALGORITHMS

As noted in Section 1, a pure parallel execution does not reduce any of the factors in Equation 2 but can be jointly applied with any of the classic cost reduction techniques, which leads to a significant decrease in time. A basic algorithm for parallel execution can split the working load into p parts, with p being the number of parallel machines, and send a part to each machine, but this distribution cannot be efficient.

One important goal with algorithms for workload distribution among processors is the optimization of the use of computational resources, on which load balance and communication traffic have a strong influence. Strategies for scheduling workloads may be classified into static and dynamic chunking [26] (a chunk is a set of tasks that are sent to a processor): whereas the static approach assigns a fixed number of tasks to each processor, the dynamic one looks for an optimal distribution of the workload to (usually) minimize the processing time. Factors to be taken into account with dynamic algorithms are the number of tasks, the number of processors, the number of chunks and the chunk size. In general, the best results are obtained with many more chunks than processors and with relatively small chunks.

In 'Guided Self-Scheduling' [27], the chunk size is calculated by dividing the number of remaining tasks between the total number of processors, thus progressively decreasing the chunk size. In 'Factoring Self-Scheduling' [28], tasks are scheduled in batches of as many chunks as processors. 'Weighted Factoring' [29] and 'Adaptive Factoring' [30] modify this with the introduction of weight factors in each processor. The chunk size is calculated by dividing the number of remaining tasks by the product of the number of processors by a parameter α ($\alpha \leq 1$). 'Trapezoid Self-Scheduling' [31] builds chunks of different sizes using a linear function: F and L tasks are respectively placed in the first and last chunks. With P being the number of processors, it seems that $F = |Tasks|/2P$ and $L = 1$ are almost optimal values. 'Quadratic Self-Scheduling' [29] determines chunk sizes on the basis of a quadratic function. There are, of course, more dynamic algorithms and different variants and combinations than those described (see, e.g., the works of Casavant and Kuhl [32] and Díaz *et al.* [26]).

For the experimental section, two static and three dynamic algorithms were implemented, which have been included in Bacterio [33], now called Bacterio$^{\text{P}}$. The tool architecture for distributed computing is presented in Section 4. An important feature of Bacterio$^{\text{P}}$ is that in practice, it uses mutant schemata [3] at byte-code level to generate the mutants. With this technique, all mutants of a class are inserted into a single file, which is instrumented with *switch* statements that help to decide what mutant to execute each time. Suppose for example that the system under test is made up of 10 classes. Then all the mutants will be distributed across 10 class files, each containing all its mutants. At the beginning (and this clarification is valid for the five algorithm implementations described in the following sections), each remote test case executor receives the 10 mutated files once. Later, during execution, Bacterio^P just requires that the *ids* of the mutants assigned in the distribution be sent to each executor.

3.1. Distribute Mutants Between Operators algorithm

Distribute Mutants Between Operators (DMBO) algorithm [16] is a static algorithm that distributes the mutants generated with a given mutation operator in as many chunks as processors (Figure 3). Thus, if there are p processors and m_{op} mutants proceeding from the *op* operator, each chunk contains m_{op}/p mutants.

Lines 2 and 3 initialize variables. Lines 4–10 create chunks with a loop that distributes, in each chunk, the same number of mutants generated with the same operator. Finally, in line 11, the chunks are sent to the processors.

This algorithm decreases the communications between nodes, and the network traffic will be minimal because each processor only receives chunks once during the mutation process. However, because of the nature of the mutants and test cases, not all executions will take the same amount of time, so presumably, and as the work by Offutt *et al.* [16] shows, some processors will finish before others, and the total process time will be the time taken by the slowest processor.

Table I describes the killing and execution matrix of a supposed system with three processors, six mutants and eight test cases. Each processor executes test cases on the mutants it has received: processor 1, for example, executes *tc1* and *tc2* on its two mutants without killing them; then it proceeds with *tc3*, which kills mutant 2, and with *tc4*, which kills mutant 1. Then no more test cases are executed on processor 1, because all its corresponding mutants have been killed. On processor 2, all test cases are executed against mutant 3, which never dies, and *tc1* kills mutant 4. On the third processor, *tc2* and *tc3* kill mutants 5 and 6.

3.2. Distribute Test Cases algorithm

Distribute Test Cases (DTC) algorithm (and adaptation to mutation testing of the load balancing strategy [34]) is another static algorithm quite similar to DMBO algorithm (Figure 4). Now, instead of splitting the mutant set, it splits the test suite into chunks depending on the number of processors:

```
1 Algorithm DMBO(m : MutantSet, ts : TestSet, p : int)   // p = number of processors
2       Let chunks be a list of p empty chunks
3       chunkIndex=1
4       for each mutation operator, op
5               Let $m_{op}$ be the list of mutants generated with op
6               for i=1 to $|m_{op}|$
7                       chunks[chunkIndex]= chunks[chunkIndex] $\cup$ {<$m_{op}$[i],ts>}
8                       chunkIndex= (chunkIndex+1)%p
9               end
10      end
11      Send each chunk in chunks to a parallel processor
12 end
```

Figure 3. Distribute Mutants Between Operators (DMBO) algorithm pseudocode.

Table I. Behaviour of Distribute Mutants Between Operators algorithm in a case with six mutants and eight test cases.

		tc1	tc2	tc3	tc4	tc5	tc6	tc7	tc8
Proc 1	Mutant 1	–	–	–	X				
	Mutant 2	–	–	X					
Proc 2	Mutant 3	–	–	–	–	–	–	–	–
	Mutant 4	X							
Proc 3	Mutant 5	–	X						
	Mutant 6	–	–	X					

```
1 Algorithm DTC(m : MutantSet, ts : TestSuite, p : int)    // p = number of processors
2         Let chunks be a list of p empty chunks
3         chunkIndex=1
4         for each test case tc in ts
5                 chunks[chunkIndex]= chunks[chunkIndex] ∪{<tc,m>}
6                 chunkIndex= (chunkIndex+1)%p
7         end
8         Send each chunk in chunks to a parallel processor
9 End
```

Figure 4. Distribute Test Cases (DTC) algorithm pseudocode.

the number of test cases, t, is distributed in as many chunks as there are processors, with t/p being test cases per chunk.

In lines 2 and 3, variables are initialized. Lines 4–7 create chunks with a loop that distributes in each chunk the same number of test cases. Finally, in line 8, the chunks are sent to the processors.

As with DMBO algorithm, the traffic and communications are minimized because each parallel processor receives only one chunk during the mutation process. However, not all executions will take the same time because each test case usually takes a different amount of time to run.

In this algorithm, there is no communication between processors, and therefore, some mutants will be killed several times in several nodes. In Table II, test cases are distributed on the three processors. Processor 1 executes *tc1* and *tc2* against mutant 3 (they do not kill it), and *tc1* kills mutant 4, which means that this processor does not attempt to kill this same mutant with *tc2*. However, because processor 2 does not know the results from processor 1, it tries to kill mutants 4 and 5 again, even though they are also killed by *tc1* and *tc5*.

3.3. Give Mutants On Demand algorithm

Give Mutants On Demand (GMOD) algorithm [17] (Figure 5) is a dynamic algorithm that splits the executions into chunks containing only one mutant with all the test cases. Thus, chunks are delivered to parallel processors several times until all the chunks are delivered.

Table II. Distribute Test Cases algorithm in the same example as Table I.

	Proc 1		Proc 2			Proc 3		
	tc1	tc2	tc3	tc4	tc5	tc6	tc7	tc8
Mutant 1	–	–	–	X		–	–	X
Mutant 2	–	–	X			X		
Mutant 3	–	–	–	–	–	–	–	–
Mutant 4	X		X			–	X	
Mutant 5	–	X	–	X		–	X	X
Mutant 6	–	–	X			X	–	–

```
1  Algorithm GMOD(m : MutantSet, ts : TestSet, ps : ProcessorsSet) //p = set of parallel processors.
2         Let chunks be a list of |m| empty chunks
3         chunkIndex=1
4         for each mutant m_i in m
5                 chunks[chunkIndex]= chunks[chunkIndex] ∪{<m_i,ts>}
6                 chunkIndex= chunkIndex+1
7         end
8         chunkProcesedIndex=1
9         while chunkProcesedIndex<chunkIndex
10                p = getAvailableProcessor(ps) //Waits until at least one processor is not executing
11                p.executeChunk(chunks[chunkProcesedIndex])
12                chunkProcesedIdex = chunkProcesedIndex+1
13        end
14 end
```

Figure 5. Give Mutants On Demand (GMOD) algorithm pseudocode.

In lines 2, 3 and 8, variables are initialized. Lines 4–7 create chunks with a loop including only one mutant in each chunk. Then in lines 9–13, chunks are sent to the parallel processors on demand. In line 10, the algorithm waits until a processor is available (it is not executing anything). When a processor is available, in line 11, a chunk is sent to it. The loop finishes when all chunks have been sent.

Unlike the DMBO and DTC algorithms, GMOD algorithm increases the communications between nodes because many chunks are sent to each processor and the network traffic is increased; thus, the network characteristics may have a strong influence on the total time. However, because of the dynamic nature of the algorithm, all the parallel processors will finish almost at the same time.

3.4. Give Test Cases On Demand algorithm

Give Test Cases On Demand (GTCOD) algorithm (an adaptation to mutation testing of the load sharing strategy described by Kapfhammer [34]) is a dynamic algorithm that splits the executions into chunks, each containing only one test case and all the mutants (see Figure 6).

In lines 2, 3 and 8, variables are initialized. Lines 4–7 create chunks with a loop including only one test case in each chunk. Then in lines 9–13, chunks are sent to the parallel processors on demand. In line 10, the algorithm waits until a processor is available. When a processor is available, in line 11, a chunk is sent to it. The loop finishes when all chunks have been sent.

Unlike the DMBO and DTC algorithms, GTCOD algorithm increases the communications between nodes because many chunks are sent to each processor and the network traffic increases. However, this algorithm increases traffic less than GMOD algorithm because the number of mutants is usually much higher than the number of test cases. Also, because of the dynamic nature of the algorithm, all the parallel processors should finish almost at the same time.

In the same way as DTC algorithm, this algorithm also performs more test case executions, because there is no communication between the processors.

3.5. Parallel Execution with Dynamic Ranking and Ordering algorithm

The Parallel Execution with Dynamic Ranking and Ordering (PEDRO) algorithm is a dynamic algorithm proposed as a novel contribution in this paper. This algorithm is based on the ideas of the ‘Factoring Self-Scheduling’ algorithm [28], but it is designed to be applied to mutations.

Parallel Execution with Dynamic Ranking and Ordering algorithm consists of two phases: (i) a ranking phase, whose behaviour is similar to DMBO algorithm, and (ii) an ordering phase similar to the GMOD algorithm. With this structure, PEDRO does not increase the network traffic (as GMOD algorithm does) thanks to the first phase and prevents some parallel processors from finishing before the others thanks to the second phase.

```
1  Algorithm GTCOD(m : MutantSet, ts : TestSuite, ps : ProcessorsSet)//p = set of parallel processors.
2      Let chunks be a list of |ts| empty chunks
3      chunkIndex=1
4      for each test case tc in ts
5          chunks[chunkIndex]= chunks[chunkIndex] ∪{<m,tc>}
6          chunkIndex= chunkIndex+1
7      end
8      chunkProcesedIndex=1
9      while chunkProcesedIndex<chunkIndex
10         p = getAvailableProcessor(ps) //Waits until at least one processor is not executing
11         p.executeChunk(chunks[chunkProcesedIndex])
12         chunkProcesedIdex = chunkProcesedIndex+1
13     end
14 end
```

Figure 6. Give Test Cases On Demand (GTCOD) pseudocode.

Both phases are configurable by two parameters, whose tuning may approximate its behaviour more to DMBO or GMOD algorithms. The first parameter is an integer fixed by the user that determines the number of chunks to be executed in each phase. The smaller the parameter, the more chunks are involved in the first phase, and vice-versa. The second parameter is the number of parallel processors that provides the required information to adapt the algorithm to the parallel system. PEDRO does not split the test suites, which avoids the problem with DTC and GTCOD algorithms related to a single mutant's many deaths.

The inputs of this algorithm are the number of parallel processors, *p*, the list of mutants, *lm*, and a numeric parameter *n* establish by the user. The pseudocode is shown in Figure 7.

Lines 2–5 initialize the variables. Lines 6–11 create the first set of chunks with a loop, distributing in each chunk the same number of mutants. Then in line 12, the first set of chunks is sent to the processors. Lines 14–23 distribute the mutants that had not been sent in different size chunks, which are created and sent to the parallel processors on demand. In line 15, the algorithm waits until a processor is available. When this happens, the size of the next chunk is calculated (line 16). Then in lines 17–21, the new chunk is created. In line 22, the chunk is sent to the available processor. The loop finishes when all mutants are sent.

As noted before, the algorithm has two phases: the ranking phase (first phase) and the ordering phase (second phase).

The ranking phase sends the same amount of work to each processor. This phase is useful to make a *ranking* of the nodes for the subsequent ordering phase. In this first phase, executions are sent to the processor nodes only once, reducing the network traffic.

Then when a processor node finishes, the ordering phase starts. In this phase, the algorithm splits the executions into chunks with a decreasing size. When a processor finishes its executions, it receives a new chunk, whose size depends on the number of remaining executions in the mutation process. Thus, when there are few remaining executions, the chunk size is small. This behaviour means that the ordering phase approximates the GMOD algorithm's behaviour at the end of the process.

This algorithm produces less traffic in the network than GMOD algorithm and prevents some parallel processors from finishing before the others and losing computational efficiency. However,

```
1 Algorithm PEDRO(m : MutantList, ts : TestSuite, p : int, α : int)//p = number of parallel processors.
2       mutantIndex = 1
3       //First phase
4       Let chunks1 be a list of p empty chunks
5       chunkSize = max( |m|/(p*α) , 1 )
6       for i=1 to |chunks1|
7               for j=1 to chunkSize
8                       chunks1[i]= chunks1[i] ∪{<m[mutantIndex],ts>}
9                       mutantIndex= mutantIndex+1
10              end
11      end
12      Send each chunk in chunks1 to a parallel processor
13      //Second phase
14      while mutantIndex <= |m|
15              p = getAvailableProcessor(ps) //Waits until at least one processor is not executing
16              chunkSize = max( (|m|-mutantIndex)/(p*α) , 1 )
17              chunk  = ∅
18              for i=1 to chunkSize
19                      chunk= chunk ∪{<m[mutantIndex],ts>}
20                      mutantIndex= mutantIndex+1
21              end
22              p.executeChunk(chunk)
23      end
24 end
```

Figure 7. Parallel Execution with Dynamic Ranking and Ordering (PEDRO) pseudocode.

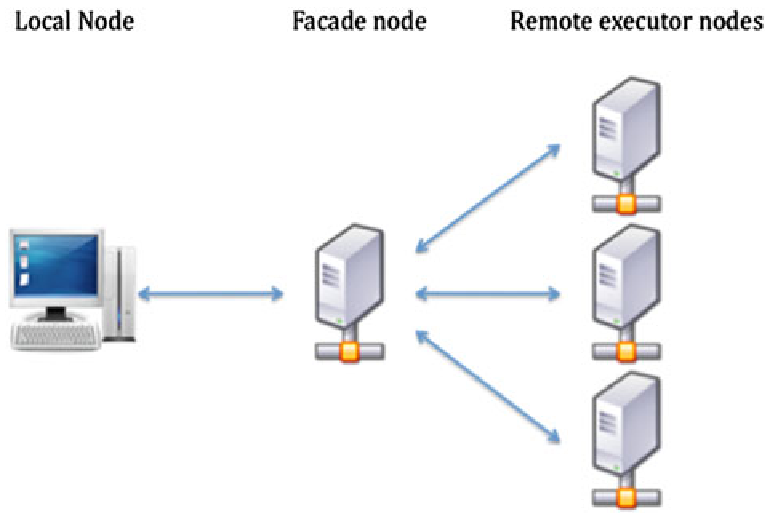


Figure 8. Parallel execution architecture.

depending on the algorithm's parameters, it can be more like the GMDO algorithm (increasing the traffic network) or more similar to DMBO algorithm (with high differences in the processor finishing times).

4. BACTERIO^P

Bacterio is a tool for mutation testing at system level developed at the University of Castilla-La Mancha. It is a commercial tool,[‡] although it is free for universities and research centres. It implements strong mutation [35], functional qualification [36], two variants of weak mutation [37] (BB-Weak/1 and BB-Weak/n) and flexible weak mutation (FWM) [33], which is specially designed to perform mutation at system level and evaluate interactions between different classes: the mutation introduced into a class A of the system may induce an anomalous state in an instance of a different class B, which maybe difficult to be detected from the test suite T_A that exercises A. FWM checks state changes in all the objects involved in the execution scenario (a more detailed description of FWM can be found in the work of Reales *et al.* [33]). Moreover, it includes selective mutation [6], high-order mutation [9], mutant sampling [4], mutant schemata [3] and byte-code translations [10].

Regarding equivalent mutants, Bacterio supports the decompilation of the Java mutated compiled code (by means of the JODE project [38]) to show the source code, although the identification of equivalent mutants must be carried out by hand. To assist in this identification, Bacterio also shows, as a result of the test cases execution, the list of mutants that have been visited and not killed: this may guide the tester to select the mutants to evaluate, so reducing the cost of equivalent mutants' detection.

The tool has been enriched with the five algorithms presented, becoming Bacterio-parallel or Bacterio^P. It can execute JUnit [39] and UISpec [40] test cases, although it is also capable of saving user interactions against the GUI in a proprietary format, which can be later reproduced to do mutation testing of the whole application from the GUI.

The architecture of Bacterio^P is depicted in Figure 8. It is made up of three kinds of nodes: (i) *local node*; (ii) *remote executor node*; and (iii) *facade node*. Each architecture node can run in a different computer.

In the *local node*, the user interacts with Bacterio to execute the supported functionalities: create mutants, execute mutants and view results. The mutant creation and the result calculation are carried out in the *local node*, and only the execution of the test cases is carried out in parallel on the

[‡]http://www.alarcosqualitycenter.com/index.php/products/bacterio

remote executor nodes. These nodes receive tests and mutants, execute them against the mutants and send back the test results. The *facade node* is in charge of receiving all the mutants and tests from the local node and of their distribution between the *remote executor nodes* based on the selected distribution algorithm.

All the communications between the nodes are implemented with Java RMI [41]. This architecture is simple and allows users to run Bacterio^P on any set of computers connected by an IP network that can run Java programs.

4.1. Local node

The local node is the previous version of the tool (Bacterio, not Bacterio^P) without the execution engine. It generates mutants and calculates the mutation scores achieved by the tests with the test results obtained from the other nodes.

4.2. Remote node

A remote node is a Java application made up of the execution engine of Bacterio and some logic to run the executions correctly. A *remoteExecutor* node really behaves as an executor server. Figure 9 shows the state machine that describes the behaviour of a *remoteExecutor* node.

Normally, this application is waiting for a mutation process to start. When a mutation process starts, this node is notified and asks for chunks (which are made up of sets of mutants and tests). When a chunk is received, the remote executor node executes the tests against the mutants that comprise the chunk and sends back the test results. Then it asks for more executions. If there are no more executions to run, it waits again for another mutation process.

4.3. Facade node

The *facade node* is connected to all the available *remote nodes* and to the *local node*. It receives all the executions to run during a mutation process and distributes them among the *remoteExecutor* nodes according to the desired algorithm (Section 3). In practice, a facade node is a distributor server. Figure 10 shows the behaviour of this node.

Normally, the *facade node* is waiting for a mutation process to start, which is notified by the *local node*. When it receives the executions of a new mutation process, it splits them into chunks (depending on the algorithm) and notifies all the *remoteExecutors* that a new mutation process has started. Then the *facade node* waits for requests of chunks, which come from the *remote nodes*. For each request, a chunk of the executions obtained previously is delivered. This process is repeated until there are no more chunks. Then it waits until all executions are finish. After the *facade node* delivers the first chunk of executions, it can receive test results at any time. In this case, the results

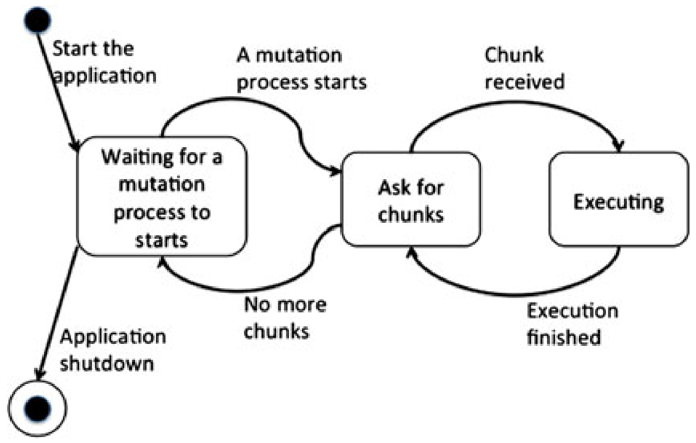


Figure 9. Behaviour of a remoteExecutor node.

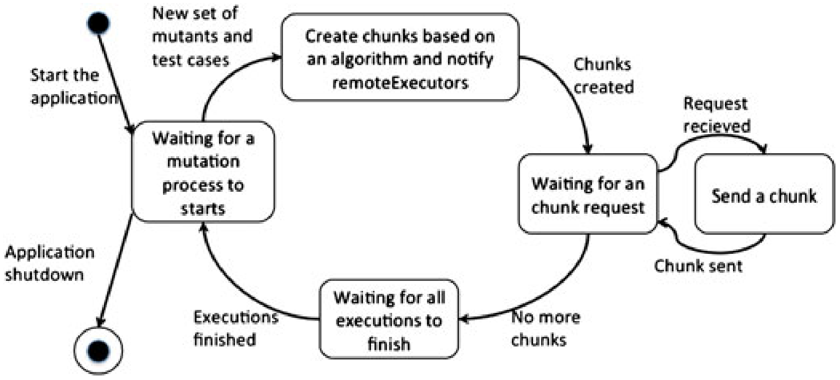

Start the application
New set of mutants and test cases
Create chunks based on an algorithm and notify remoteExecutors
Chunks created
Request recieved
Waiting for a mutation process to starts
Waiting for an chunk request
Send a chunk
Chunk sent
Application shutdown
Executions finished
Waiting for all executions to finish
No more chunks

Figure 10. Behaviour of the facade node.

are sent back to the local node, which updates the user interfaces and calculates the mutation score. When all the executions are performed, the *facade node* stops and waits again for another mutation process.

4.4. Notes about the general process

Two of the reduction cost techniques implemented in Bacterio have a direct impact on the parallel execution process when a mutation analysis is performed with Bacterio^P.

- Bacterio generates mutants with *byte-code translations* [10]; thus, mutants are directly executable. They do not have to be compiled; therefore, the *remoteExecutors* only have to run the tests against the mutants.
- Bacterio implements *mutant schemata* [3]. The mutant schemata technique creates all the mutants and the original system in the same code. This means that after the generation process, there will only be one copy of each file comprising the system under test, which contains the code of all the mutants. The code of each mutant is instrumented with flags controlled by Bacterio, and it is only executed when a mutant must be executed; otherwise, the original code is executed.

This has two advantages: no mutant compilation is required, and only one copy of the system under test exists. This means a reduction in the communications between nodes, because each *remoteExecutor* node only needs to receive a compiled copy of the system to execute any mutant. Thus, when a *remoteExecutor* node asks for executions, the *facade node* only sends it the ids of the mutants and the names of the test cases to run.

Table III. Goal–Question–Metric template.

Goal	To investigate the adequacy of different distribution algorithms in different environments to split the execution phase of the mutation analysis process.
Question	Which of the distribution algorithms works best for each environment during the execution mutant phase?
Metric	The total time of the execution mutant process measured in seconds, the communication time of the parallel nodes that execute the mutants measured in seconds, the waiting time of the parallel nodes that execute the mutants and the total number of executions.
Object of study	Distribution algorithms: DMBO, DTC, GMOD, GTCOD and PEDRO.
Purpose	To evaluate and improve the effectiveness of mutation testing using parallel execution techniques.
Focus	To investigate the adequacy of different distribution algorithms.
Perspective	From the point of view of the researcher.
Context	In different environments (Section 5.2).

DMBO, Distribute Mutants Between Operators; DTC, Distribute Test Cases; GMOD, Give Mutants On Demand; GTCOD, Give Test Cases On Demand; PEDRO, Parallel Execution with Dynamic Ranking and Ordering.

Table IV. Computer characteristics for the 8 + 1 heterogeneous environment.

Id	Node	Processor	P_Speed (GHz)	P_Cores	Memory (GB)	Operating system
PC0	Local and facade	Intel Core 2 Duo	2.66	2	4	Mac OS X 10.6.6
PC1	Remote Executor	Intel Pentium 4	3.40	2	1	Windows XP Pro SP3
PC2		Intel Pentium 4	3.40	2	2	Windows XP Pro SP3
PC3		AMD Sempron 3000+	1.81	1	2	Windows XP Pro SP3
PC4		AMD Athlon 64 ×2 Dual Core 4200+	2.21	2	2	Windows XP Pro SP3
PC5		Intel Pentium 4	3.40	2	2	Windows 7 Pro
PC6		Intel Pentium 4	3.40	2	2	Windows XP Pro SP3
PC7		Intel Pentium 4	3.40	2	2	Windows XP Pro SP3
PC8		AMD Athlon 64 ×2 Dual Core 4200+	2.21	2	2	Windows XP Pro SP3

This is important for the study presented in this paper because regardless of the algorithm used to distribute the executions, the time expended to send the system under test and the mutants will be the same.

5. EXPERIMENT DESIGN

As noted in Section 2, there is a need to study how to split the execution phase of a mutation analysis to improve the efficiency of the mutation. The experiments presented in this section study this issue. Some configuration environments to parallelize the mutation process were selected, and different algorithms were used to split the execution phase. Then a mutation analysis of different test suites of some applications was performed. The results show the most suitable algorithms (among those implemented) for specific environments with the current technologies.

5.1. Research goals

Table III shows the goals of this experiment using the Goal–Question–Metric template [42].

5.2. Environment description

The algorithms were tested on six environments with different characteristics, three of them homogeneous (all computers have the same processor, memory and operating system) and three heterogeneous. Moreover, their configurations were modified to give the environments a variable number of computers and networks.

5.2.1. 8 + 1 *heterogeneous computers.* The smallest environment selected was made up of eight heterogeneous computers for running the *remoteExecutors* nodes and one to run the facade and local nodes. The computers were provided by Alarcos research group [43]. They were common desktop computers used by researchers. Table IV shows their characteristics.

The network for this environment had the configuration shown in Figure 11. The network speed in all the links was 100 Mbits/s.

5.2.2. 19 + 1 *homogeneous environment.* This environment was made up of 19 homogeneous computers to run the *remoteExecutors* nodes and a different computer to run both the facade and local nodes. The *remoteExecutors* were the computers in a laboratory used by students. Each one had a Fedora Linux operating system with a VMWare virtual machine to run the configuration suitable for all of the subjects taught at the School of Computer Science in the University of Castilla-La Mancha. For the experiments, the VMWare system was launched. Windows 2000 was loaded in every 512 Mb computer (Table V).

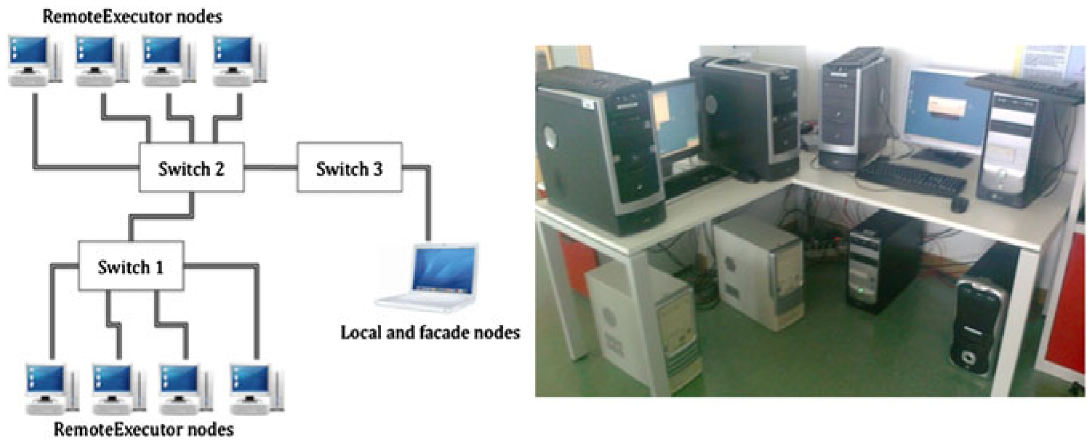


Figure 11. Network configuration of the 8 + 1 heterogeneous environment and a picture.

Table V. Computer characteristics for the 19 + 1 homogeneous environment.

Id	Node	Processor	P_Speed (GHz)	P_Cores	Memory	Operating system
PC0	Local and facade	Intel Core 2 Duo	2.66	2	4 GB	Mac OS X 10.6.6
PC1 to PC19	Remote Executor	Intel Core 2 Duo E7500	2.93	2	512 Mb	Windows 2000

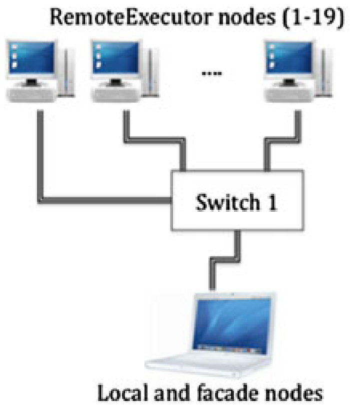


Figure 12. Network configuration of the 19 + 1 heterogeneous environment and a picture.

Table VI. Computer characteristics for the 40 + 1 homogeneous environment.

Id	Node	Processor	P_Speed (MHz)	P_Cores	Memory (GB)	Operating system
PC0	Local and facade	2 × AMD Opteron QuadCore	2300	8	16	Linux Rocks
PC1 to PC10	4 × Remote Executor	2 × AMD Opteron QuadCore	2300	8	16	Linux Rocks

The network speed was 100 Mbits/s. Its configuration and a picture is shown in Figure 12.

5.2.3. 40 + 1 *homogeneous environment.* Eleven homogeneous computers made up this environment, 10 to run 40 *remoteExecutor* nodes and one to run the local and facade nodes. These computers are part of the Hermes cluster kindly provided by the University of Castilla-La Mancha QCyCAR research group [44]. All of these computers have two processors with four cores each. Four *remoteExecutors* were run in each computer to exploit their efficiency. Table VI shows the characteristics of these computers.

The network for this environment was made up of optic fibre links with two speeds: 1 and 10 Gbit/s (Figure 13).

5.2.4. 40 + 1 *heterogeneous environment.* Twenty-one computers were selected for this environment, 20 of them ran 40 *remoteExecutor* nodes and one ran the local and facade nodes. These computers also belong to the Hermes cluster kindly provided by the CQyCAR research group.

In this environment (Table VII), there were four kinds of computers: one had six computers, whereas the other three had five. The local and facade nodes ran on another computer.

The network speeds were similar to the 40 + 1 homogeneous environment, but some computers were placed in different raids; thus, the configuration was different (Figure 14).

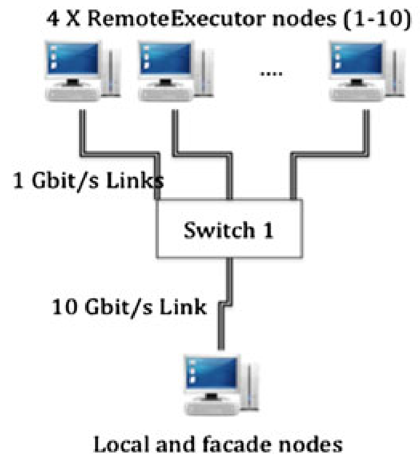


Figure 13. Network configuration for the 40 + 1 homogeneous environment and a picture of the Hermes cluster.

Table VII. Computer characteristics for the 40 + 1 heterogeneous environment.

Id	Node	Processor	P_Speed (MHz)	P_Cores	Memory (GB)	Operating system
PC0	Local and facade	2 × AMD Opteron QuadCore	2300	8	16	Linux Rocks
PC1 to PC5	4 × Remote Executor	2 × AMD Opteron QuadCore	2300	8	16	Linux Rocks
PC6 to PC10	2 × Remote Executor	AMD Opteron QuadCore	2200	4	8	Linux Rocks
PC11 to PC15	1 × Remote Executor	AMD Opteron Dual Core	2800	2	2	Linux Rocks
PC16 to PC20	1 × Remote Executor	Intel Xeon	3200	1	2	Linux Rocks

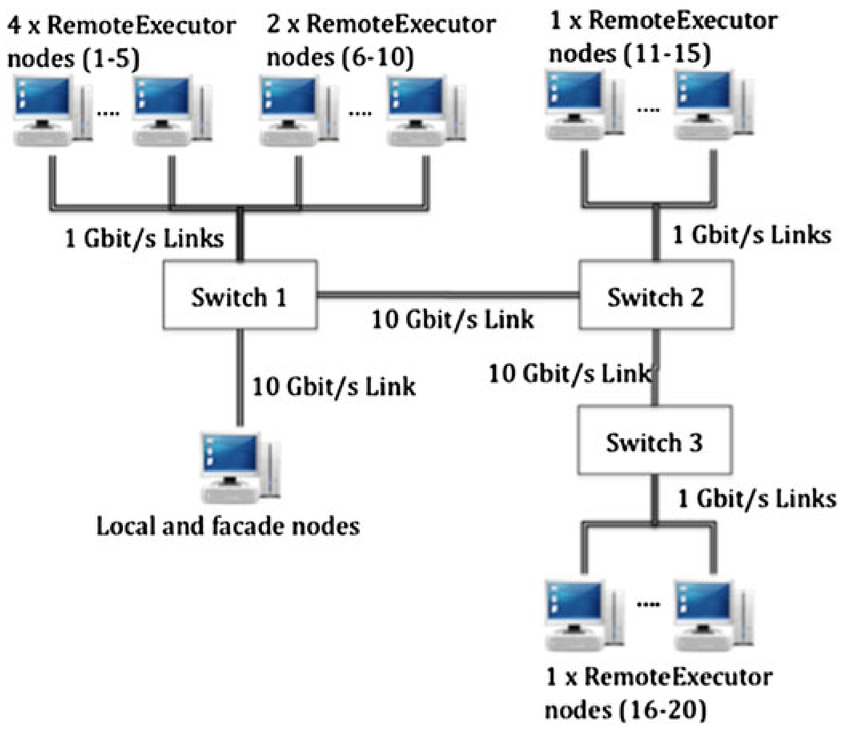


Figure 14. Network configuration for the 40 + 1 heterogeneous environment.

Table VIII. Computer characteristics for the 80 + 1 heterogeneous environment.

Id	Node	Processor	P_Speed (MHz)	P_Cores	Memory (GB)	Operating system
PC0	Local and facade	2 × AMD Opteron QuadCore	2300	8	16	Linux Rocks
PC1 to PC10	8 × Remote Executor	2 × AMD Opteron QuadCore	2300	8	16	Linux Rocks

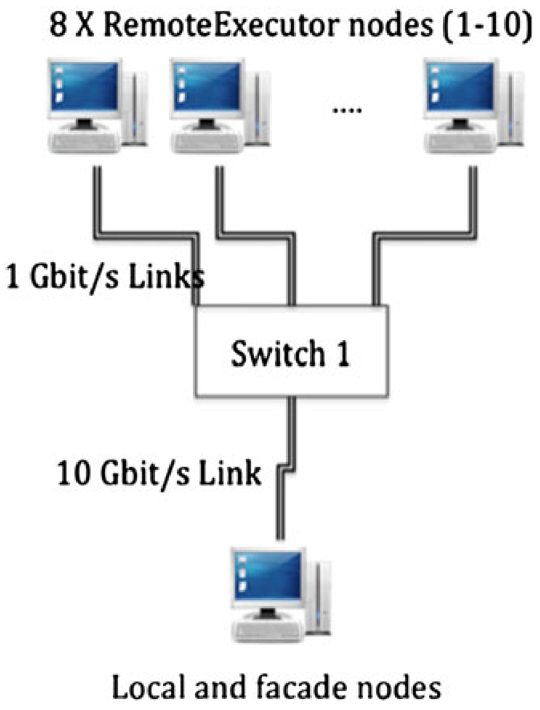


Figure 15. Network configuration for the 80 + 1 homogeneous environment.

5.2.5. 80 + 1 *homogeneous environment.* This environment was similar to the 40 + 1 environment, with the difference that each computer ran twice as many *remoteExecutor* nodes to completely load the system and exploit all the available processor cores. Thus, 11 computers were selected, 10 to run 80 *remoteExecutor* nodes and one to run the local and facade nodes. This computer had two processors with four cores each; thus, to use all the cores, eight *remoteExecutor* nodes were run in each computer. These computers are also part of the Hermes cluster kindly provided by the CQyCAR research group. Table VIII shows the characteristics of the selected computers.

The network configuration was similar to the 40 + 1 homogeneous environment. The only difference was that here there were eight *remoteExecutor* nodes running on each computer. Figure 15 shows this network configuration.

5.2.6. 80 + 1 *heterogeneous environment.* This was the last environment used in the experiments. It was similar to the 40 + 1 heterogeneous environment but with some differences: there were 26 computers and twice as many *remoteExecutors* nodes running on each. Twenty-five computers were used to run 80 *remoteExecutor* nodes and one to run the *local* and *facade* nodes.

The environment had four kinds of computers (Table IX).

The network configuration was also similar to the 40 + 1 heterogeneous environment. Figure 16 shows this configuration.

Table IX. Computer characteristics for the 80 + 1 heterogeneous environment.

Id	Node	Processor	P_Speed (MHz)	P_Cores	Memory (GB)	Operating system
PC0	Local and facade	2 × AMD Opteron QuadCore	2300	8	16	Linux Rocks
PC1 to PC5	8 × Remote Executor	2 × AMD Opteron QuadCore	2300	8	16	Linux Rocks
PC6 to PC10	4 × Remote Executor	AMD Opteron QuadCore	2200	4	8	Linux Rocks
PC11 to PC15	2 × Remote Executor	AMD Opteron Dual Core	2800	2	2	Linux Rocks
PC16 to PC25	1 × Remote Executor	Intel Xeon	3200	1	2	Linux Rocks

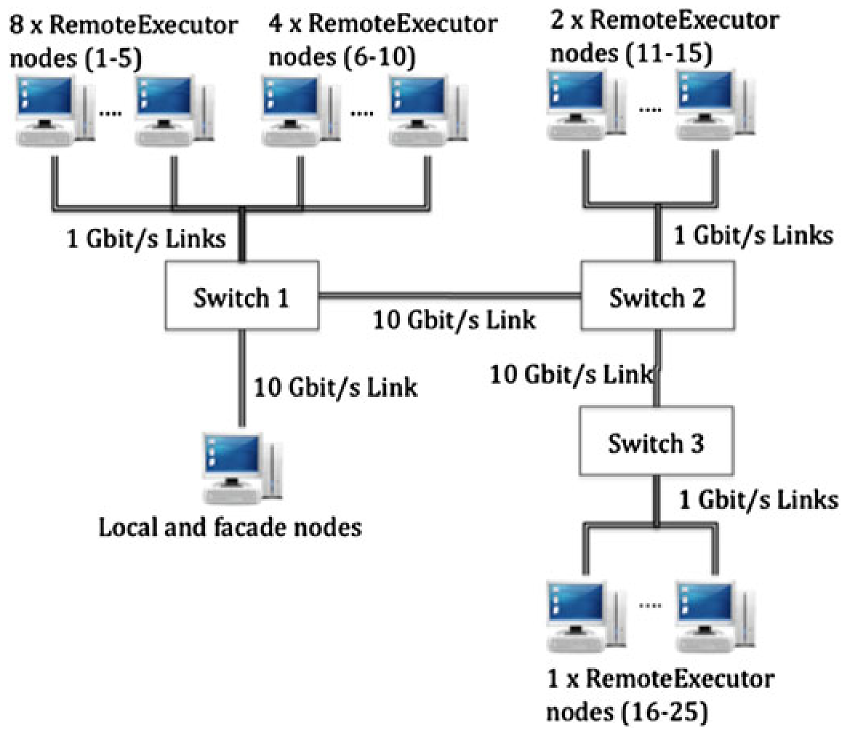


Figure 16. Network configuration for the 40 + 1 heterogeneous environment.

5.3. *Applications under tests*

The experiments were run on four applications. Three of them were used to compare the five algorithms described in Section 3. Then the two most similar algorithms were compared with the fourth application. All the applications were written in Java. The applications were as follows:

1. *Monopoly*. This application was implemented by professors of Software Engineering II (a fifth year Computer Science studies subject at the University of Castilla-La Mancha). Students use this application to check their skills in test writing. This application simulates the Monopoly board game.
2. *Cinema*. This is a single application implemented for the purposes of software engineering experimentation. It is a cinema management system with a facility to reserve locations.
3. *Chess*. This application was written by fifth year Computer Science students. It simulates an online chess game to play on a network against other human players. Only the server part of the applications was used in the experiments.

Table X. Quantitative description of the applications.

	Metrics of the systems under test					Test cases		
Application	Lines of code	Classes	Methods	Average complexity	Maximum complexity	Number	Instructions coverage (%)	Execution time (s)
Monopoly	654	40	135	2.044	10	108	95.08	0.59
Cinema	678	10	106	2.896	58	140	83.41	0.31
Chess	2288	13	108	7.917	145	137	70.48	0.90
Analysis	2398	34	195	2.115	20	152	89.37	0.78

4. *Analysis package of Commons Math* [13]. This application is a library of classes with mathematics and statistics functionalities. It is a free and publicly available package, released under the Apache License, and created to address the most common mathematical problems not available in the Java programming language.

Each application had a test suite in JUnit format. These test suites were used to perform mutation analysis and obtain the results of the experiments. The applications Monopoly, Cinema and Chess were used to compare the five algorithms (Section 3). Then the Analysis package was used to compare the two algorithms that offered the most similar results.

5.3.1. Quantitative description of applications and test suites. Table X quantitatively describes the applications with some metrics, measured with the Eclipse Metrics 1.3.6 plug-in [45]: number of lines of code, classes and methods, and average and maximum McCabe cyclomatic complexity of the methods. The test suites are described through the tests numbers, the instructions coverage (measured with the EclEmma plug-in for Eclipse [46]) achieved by the tests and the total time to run all the tests (measured in a laptop with Windows 7 and an Intel Core 2 Duo P8400 2.26 GHz).

Although the described applications cannot be considered large systems, they are large enough to show significant differences between the workload distribution algorithms that will be studied in this work.

5.4. Variable description

The required time for each mutation process at each algorithm and environment was measured to compare the efficiency of the distribution algorithms in different environments. Also, because the environments had different configurations (network, processor speed, memory and homogeneity or heterogeneity), the communication time between the *remoteExecutor* nodes and the *facade node* was measured, as well as the waiting time for each processor until the accomplishment of the whole process.

There were three *dependent variables* in the experiments. For the whole process, the *total time* was measured (i.e. the time spent from the moment the user orders the execution until all the *remoteExecutors* finish their job). For each *remoteExecutor*, the times collected were the *communication time* (time spent for communication between the *remoteExecutor* node and the *facade node*) and the *waiting time* (time since the *remoteExecutor* node ends its job until the whole mutation process is finished). Finally, the *total number of executions* (sum of the number of 'mutant-test case' pairs executed by each *remoteExecutor*) was also measured.

There were two *independent variables*: (i) *distribution algorithm*, which had five values (DMBO, DTC, GMOD, GTCOD, PEDRO), and (ii) *environment*, which had six values.

Note that different values of the *environment* variable can be used to do different studies: homogeneous and heterogeneous environments, small and big environments, and fast network and slow network environments. However, depending on the study, the variables must be analysed in different ways.

5.5. *Tools*

Some additional tools were used to deal with the Java applications used in the experiments: Eclipse [47] as integrated development environment with the Eclipse Metrics 1.3.6 [45] and EclEmma [46] plug-ins. Additionally, the JUnit framework [39] was used because the test suites are in this format.

The tool used to perform the mutation analysis with parallel executions was Bacterio^P (Section 4). The configuration of Bacterio^P was the same for all the experiments:

- Mutation operators: SWAP (interchanges two type-compatible arguments of a method), TEX (throws an exception when a method is called), INC (increments the value of a numeric argument of a method), DEC (decrements the value of a numeric argument of a method), NUL (sets to null an object argument of a method), AOR (replaces an arithmetic operator by other), ROR (replaces a relational connector by other), ABS (sets the absolute value in a value assignment), UOI (inserts an unary operator) and LCR (replaces a logical operator by other).
- Mutation analysis technique: FWM technique [33] that is implemented in Bacterio^P and was specially designed to perform mutation analysis at system level.
- FWM needs a method to analyse the state of a concrete object during the original and mutant executions to kill mutants. The method selected was the *Hashcode* method implemented in Bacterio^P, with a depth of 3, which is based in the calculus of the object hashcode.
- Higher-order mutation: first-order mutation (higher-order mutants were not used).
- Timeout: 3 s.

For the distribution algorithms, only the PEDRO algorithm needed a configuration parameter (Section 3.5). This parameter was established as the number of *remoteExecutor* nodes.

Finally, to manage and statistically study the data collected from the experiments, the SPSS application and Microsoft Excel to store and manipulate the data were used.

5.6. *Experimental procedure and statistics*

The following subsections describe all the procedures carried out to obtain and analyse the experimental data.

5.6.1. Obtaining times with a specific environment, algorithm and application. The procedure to obtain times with the specific values of the independent variables forms the base of the experiments. The following paragraphs describe this procedure. The procedure inputs are the environment *ev*, the application *ap* and the algorithm *da*.

1. Deploy the Bacterio^P application in the environment *ev*.
   1.1. Run all the *remoteExecutor* nodes in the computers following the specifications of the environment *ev*.
   1.2. Run the *facade node* in the computer established by the environment *ev*.
   1.3. Configure the *facade node* including all the ip:port addresses of each *remoteExecutor*.
   1.4. Configure the *facade node* to select the distribution algorithm *da*.
   1.5. Run the *local node*.
2. Configure the *local node* to work with the *ap* application.
3. Configure the *local node* with the configuration established for the experiments.
4. Create four empty sheets in an Excel file.
5. Generate mutants.
6. Execute all tests against the mutants.
7. Import the time data file created by the facade node into Excel sheet number 1.
8. Initialize the application under test and the testing environment.
9. Execute the mutants against all the tests the second time.
10. Import the times data file created by the facade node into Excel sheet number 2.
11. Initialize the application under test and the testing environment.
12. Execute the mutants against all the tests the third time.
13. Import the times data file created by the facade node into Excel sheet number 3.

14. Create Excel page number 4.
15. Make the mean of all the values from pages 1 to 3 and store it in the fourth Excel sheet.

At the end of this process, there is one Excel sheet with the time information obtained by Bacterio^{P}. Because the time data are not completely deterministic (because the execution time can be affected by several elements, such as the operating system scheduler, interruptions, etc.), the times are collected three times, and their mean is calculated.

Note that equivalent mutants are neither identified nor excluded from the experiments, because this would cloud the results: in fact, knowing what mutants are equivalent requires the previous execution of all the mutants (which is one of the goals in this article) and, then, to search the equivalent ones among the alive ones. Furthermore, the goal of the experiment is not the determination of mutations scores but the goodness of different parallel execution algorithms.

The information obtained by the facade node is as follows:

- Distribution algorithm.
- Total time of the execution phase of the mutation process.
- Number of ordered chunks.
- Number of *remoteExecutor* nodes.
- For each *remoteExecutor* node:
  - Remote executor total process time (time spent by the remote executor to execute all the chunks supplied by the *facade node*).
  - Communication time (time spent in communication between the *remoteExecutor* node and the *facade node*).
  - Waiting time (time from when the *remoteExecutor* finishes until the global process finishes).

To perform the experiments, the described procedure was carried out with several combinations of the values of the independent variables in two phases: In the first phase, the procedure was performed with all the combinations of the values shown in Table XI. With this, all the combination algorithms were compared in all the environments with three applications (Monopoly, Cinema and Chess). This required the execution of $6*5*3 = 90$ combinations.

After performing the procedure with all the combinations, the obtained data showed that the PEDRO and GMOD algorithms are very similar. Thus, in a second phase, these two algorithms were executed with the fourth application (the Apache Analysis package). This allowed to obtain additional data and to perform an improved statistical analysis. The data obtained from this second phase were useful to determine whether or not the algorithms were similar. The combinations used are those in Table XII.

One of the reasons for this experimental set up (comparison of the algorithms in two phases) was the lack of complete availability of the cluster, only usable during short periods. So, the second phase focused on PEDRO and GMOD algorithms, because the results of phase 1 showed scarce differences between them.

Table XI. Combinations to compare all distribution algorithms.

	Environment	Algorithm	Application
	8 + 1 heterogeneous	PEDRO	Monopoly
	19 + 1 homogeneous	GMOD	Cinema
	40 + 1 homogeneous	GTCOD	Chess
	40 + 1 heterogeneous	DMBO	
	80 + 1 homogeneous	DTC	
	80 + 1 heterogeneous		
Total values	6	5	3

PEDRO, Parallel Execution with Dynamic Ranking and Ordering; GMOD, Give Mutants On Demand; GTCOD, Give Test Cases On Demand; DMBO, Distribute Mutants Between Operators; DTC, Distribute Test Cases.

Table XII. Combinations to compare PEDRO and GMOD algorithms.

Comb. id	Environment	Algorithm	Application
1	8 + 1 heterogeneous	PEDRO	Analysis package
2	8 + 1 heterogeneous	GMOD	Analysis package
3	19 + 1 homogeneous	PEDRO	Analysis package
4	19 + 1 homogeneous	GMOD	Analysis package
5	40 + 1 homogeneous	PEDRO	Analysis package
6	40 + 1 homogeneous	GMOD	Analysis package
7	40 + 1 heterogeneous	PEDRO	Analysis package
8	40 + 1 heterogeneous	GMOD	Analysis package
9	80 + 1 homogeneous	PEDRO	Analysis package
10	80 + 1 homogeneous	GMOD	Analysis package
11	80 + 1 heterogeneous	PEDRO	Analysis package
12	80 + 1 heterogeneous	GMOD	Analysis package

PEDRO, Parallel Execution with Dynamic Ranking and Ordering; GMOD, Give Mutants On Demand.

5.6.2. Statistical analysis. To analyse the data and establish conclusions, some statistical tests were selected. The statistical tests mitigated the generalization threads and increased the validity of the obtained conclusions.

The data collected do not follow a normal distribution, what avoided the application of the ANOVA test. So, two non-parametric tests, which do not require preconditioning the data, were selected. The Friedman test [48] was selected to compare the times of the distribution algorithms. This test is designed to detect differences in treatments across multiple test attempts for complete block designs. In the experiments here, there is a complete block design where the different treatments are the distribution algorithms. Thus, this test can determine if there are significant differences between the times obtained with each algorithm.

To compare the two most efficient algorithms (PEDRO and GMOD), the Wilcoxon signed-rank test [49], which is designed to compare two related samples and assess whether their population means differ, was selected. In the experiments, the related data are the time obtained with the PEDRO and GMOD algorithms in a specific environment and with a specific application. Thus, this test can determine whether there are significant differences between the times obtained with PEDRO and GMOD algorithms.

The third comparison carried out in the experiments was related to homogeneous and heterogeneous environments. To compare the times obtained in these two kinds of environments, the Wilcoxon signed-rank test was also applied. In this case, the related data were the times obtained in a homogenous environment and in a heterogeneous environment with the same distribution algorithm, application and number of parallel nodes. The Wilcoxon signed-rank test can determine if there are significant differences between the times obtained in these two kinds of environments.

As three different statistical tests were performed, there are three different sets of data points:

- Ninety data points were collected, for the comparison of the five distribution algorithms.
- Thirty-two data points, for comparing PEDRO and GMOD algorithms.
- Sixty data points, to compare the homogenous and heterogeneous environments.

Each data point represents the total execution time of one mutation analysis with a combination ‘application-environment-distribution algorithm’.

6. EXPERIMENT RESULTS AND DISCUSSION

This section presents the results obtained with each algorithm and the statistical analysis results. Table XIII shows the number of mutants generated from each application.

Table XIII. Number of generated mutants.

Application	Number of generated mutants
Cine	2040
Monopoly	1250
Chess	5737
Analysis	8036

6.1. Comparison of all the algorithms in each environment and application

The total times for the execution phase of the mutation process are shown in Figure 17, which has six charts, one for each environment. Each shows the total execution time in seconds for each application with each distribution algorithm. The chart also has a table with the specific values.

As the charts in Figure 17 show, PEDRO and GMOD algorithms obtained the best results. Their times were very similar, except in two cases: PEDRO algorithm obtained better results in the 8 + 1 heterogeneous environment with *Cine* and *Chess* and in the 19 + 1 homogeneous environment; PEDRO algorithm also obtained better results with the Chess application (between 7 and 32 s faster than the GMOD algorithm). This leads to think that these two algorithms are very similar, and the study was thus extended with a fourth application to compare them. The results of this study are shown in Section 6.2.

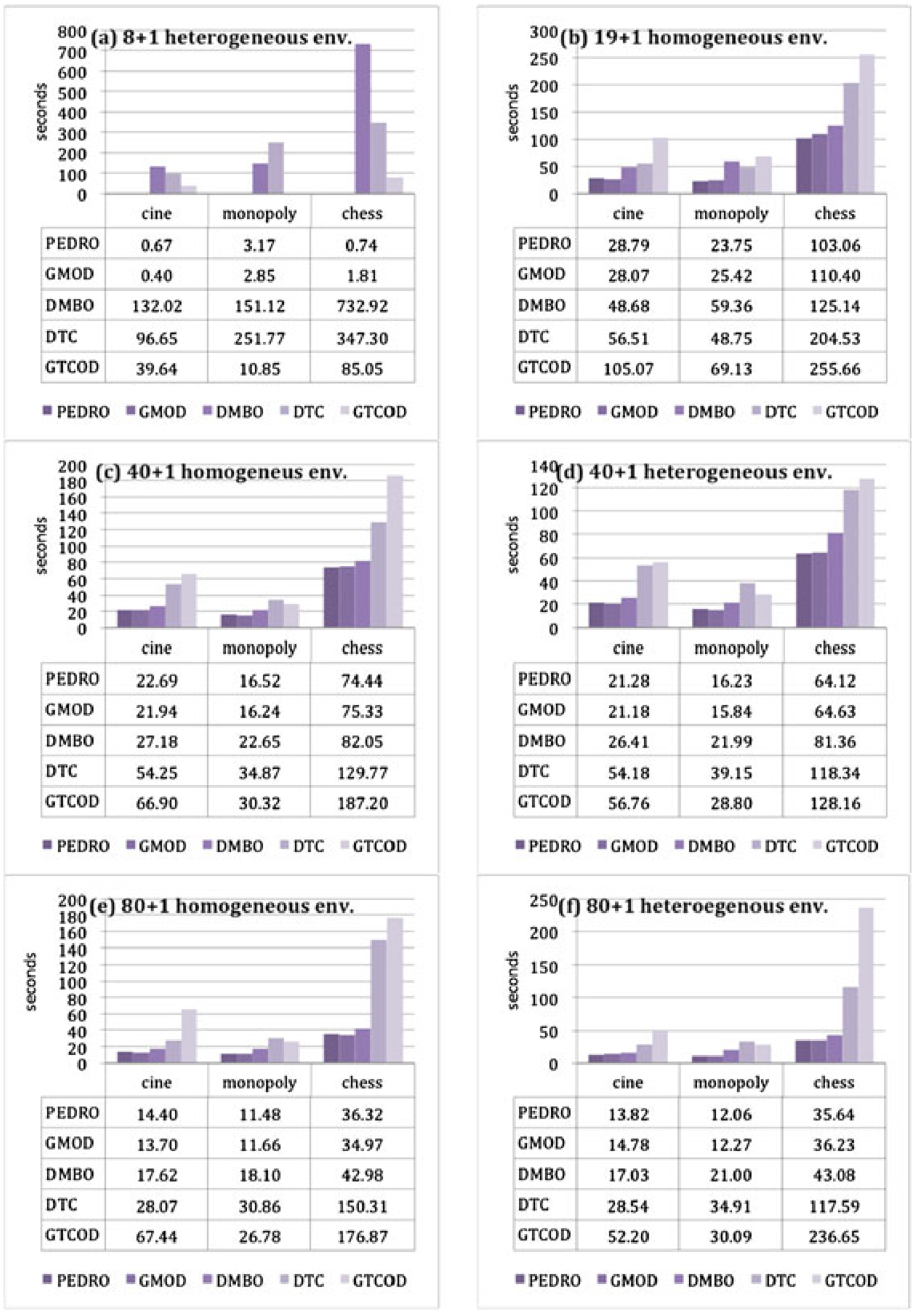
With this data, the third best algorithm is DMBO. In the 40 + 1 and 80 + 1 environments, the DMBO algorithm also obtained slightly worse results than PEDRO and GMOD algorithms (between 4 and 16 s slower). Note that with smaller environments, the algorithm becomes worse. These differences are due to the waiting times.

Figure 18 shows the average waiting time (time from when the *remoteExecutor* finished to when the overall process finished) for each node in a specific environment with each application and algorithm. As Figure 18 shows, with the 8+1 heterogeneous and 19+1 homogeneous environments, the average waiting time for the nodes using the DMBO algorithms is too long (between 11 and 732 s) compared with the average waiting time obtained with the PEDRO or GMOD algorithm (between 0 and 3 s). However, in the 40 + 1 and 80 + 1 environments, the average waiting times for DMBO algorithm are lower (between 3 and 15 s), which makes DMBO algorithm's total times more similar to the total times obtained with PEDRO and GMOD algorithms in these environments.

Another reason for these differences is the influence of the communication time. Figure 19 shows the average time that each *remoteExecutor* node in a specific environment spent in communications with the facade node. These charts show that as theoretically expected, the GMOD algorithm required more communication time. However, this wasted time is much lower than the waiting time wasted by DMBO algorithm. Also, the percentage of the communication time with respect to the total time was very small, and thus, it had almost no influence on the results: for example, in the 40 + 1 environments for the GMOD algorithm, the communication time was between 0 and 2 s, but the total time was between 82 and 21 s. Therefore, the waiting time had almost no influence on the total time, which means that the algorithm that reduces the total time at the expense of waiting time (GMOD and PEDRO algorithms) is better than the algorithms that do the opposite (DMBO algorithm).

A special case is the communication time obtained in the 80 + 1 environments with the Chess application. In these two cases, the DMBO algorithm obtained the worst results (between 8 and 6 s), when theoretically the GMOD algorithm should be the worst algorithm because there are more communications with the facade node (in fact, the rest of charts shows that GMOD algorithm obtained the worst communication times). With the DMBO algorithm, the mutants are very dispersed, and the characteristics of the application mean that the time to run all the test cases with these subsets of mutants is very similar. Therefore, many *remoteExecutor*s must finish at the same time, and collisions occur when giving the test results to the facade node.

Finally, the worst algorithms are DTC and GTCOD. Although DTC algorithm obtained better results than GTCOD algorithm, both of them obtained much worse results than the other three algorithms. There are two reasons for this.



(a) 8+1 heterogeneous env.

	cine	monopoly	chess
PEDRO	0.67	3.17	0.74
GMOD	0.40	2.85	1.81
DMBO	132.02	151.12	732.92
DTC	96.65	251.77	347.30
GTCOD	39.64	10.85	85.05

(b) 19+1 homogeneous env.

	cine	monopoly	chess
PEDRO	28.79	23.75	103.06
GMOD	28.07	25.42	110.40
DMBO	48.68	59.36	125.14
DTC	56.51	48.75	204.53
GTCOD	105.07	69.13	255.66

(c) 40+1 homogeneus env.

	cine	monopoly	chess
PEDRO	22.69	16.52	74.44
GMOD	21.94	16.24	75.33
DMBO	27.18	22.65	82.05
DTC	54.25	34.87	129.77
GTCOD	66.90	30.32	187.20

(d) 40+1 heterogeneous env.

	cine	monopoly	chess
PEDRO	21.28	16.23	64.12
GMOD	21.18	15.84	64.63
DMBO	26.41	21.99	81.36
DTC	54.18	39.15	118.34
GTCOD	56.76	28.80	128.16

(e) 80+1 homogeneous env.

	cine	monopoly	chess
PEDRO	14.40	11.48	36.32
GMOD	13.70	11.66	34.97
DMBO	17.62	18.10	42.98
DTC	28.07	30.86	150.31
GTCOD	67.44	26.78	176.87

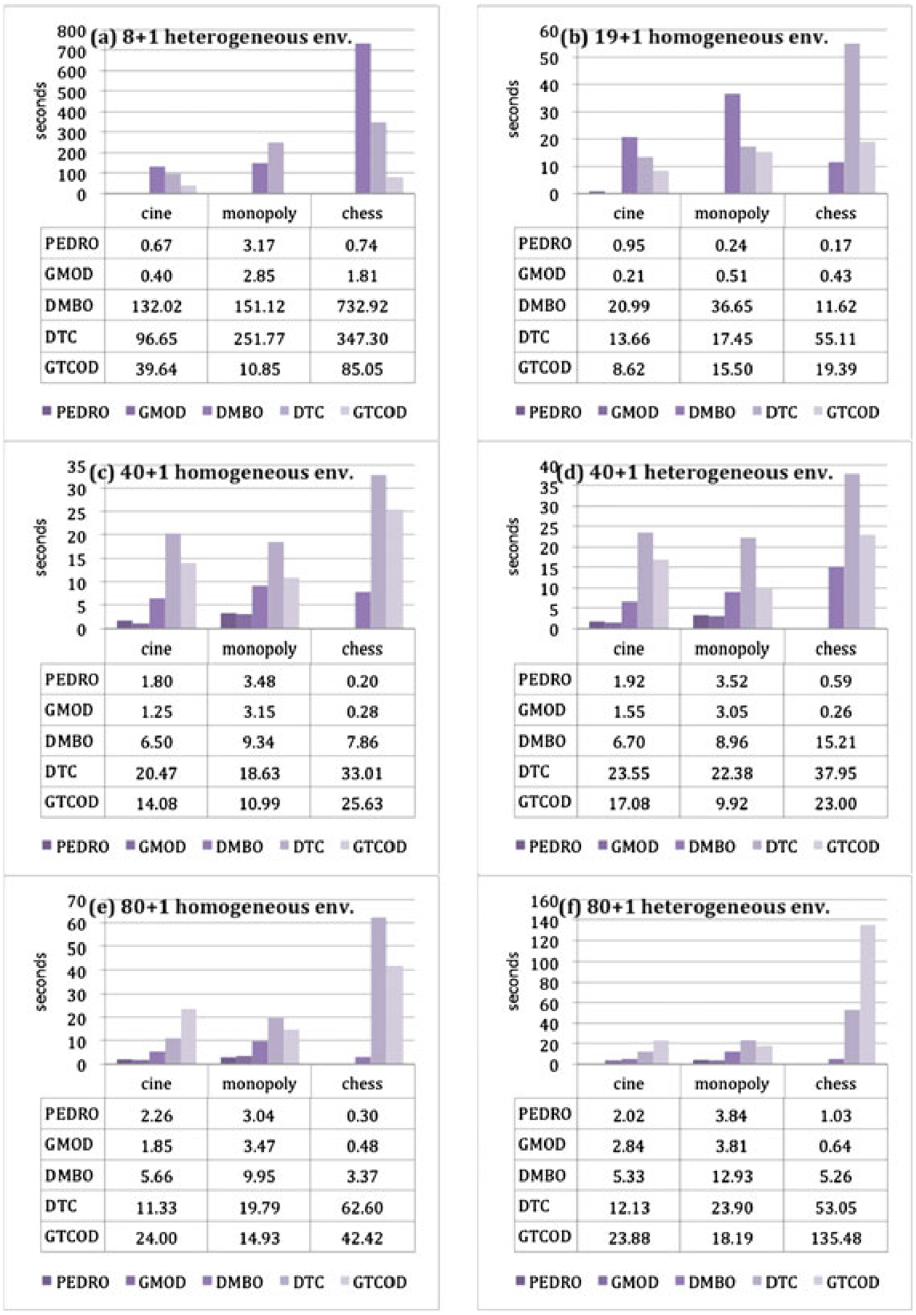
(f) 80+1 heteroegenous env.

	cine	monopoly	chess
PEDRO	13.82	12.06	35.64
GMOD	14.78	12.27	36.23
DMBO	17.03	21.00	43.08
DTC	28.54	34.91	117.59
GTCOD	52.20	30.09	236.65

Figure 17. Total times with Cinema, Monopoly and Chess.

First, these algorithms obtained the worst waiting times, as Figure 18 shows. In the case of the DTC algorithm, the waiting times were longer than with the DMBO algorithm (which splits the execution work into the same number of chunks and the waiting time could be the same) because splitting the test suites makes the parts very different. This means that some *remoteExecutor* nodes can finish very early whereas others finish late.

In the case of GTCOD algorithm, the waiting times are longer than with the GMOD algorithm (whose waiting time could be the same) because each time that a *remoteExecutor* node asks for more executions, it receives a much larger number of executions than with the GMOD algorithm, because the number of executions depends on the number of mutants instead of the number of test



(a) 8+1 heterogeneous env.

	cine	monopoly	chess
PEDRO	0.67	3.17	0.74
GMOD	0.40	2.85	1.81
DMBO	132.02	151.12	732.92
DTC	96.65	251.77	347.30
GTCOD	39.64	10.85	85.05

(b) 19+1 homogeneous env.

	cine	monopoly	chess
PEDRO	0.95	0.24	0.17
GMOD	0.21	0.51	0.43
DMBO	20.99	36.65	11.62
DTC	13.66	17.45	55.11
GTCOD	8.62	15.50	19.39

(c) 40+1 homogeneous env.

	cine	monopoly	chess
PEDRO	1.80	3.48	0.20
GMOD	1.25	3.15	0.28
DMBO	6.50	9.34	7.86
DTC	20.47	18.63	33.01
GTCOD	14.08	10.99	25.63

(d) 40+1 heterogeneous env.

	cine	monopoly	chess
PEDRO	1.92	3.52	0.59
GMOD	1.55	3.05	0.26
DMBO	6.70	8.96	15.21
DTC	23.55	22.38	37.95
GTCOD	17.08	9.92	23.00

(e) 80+1 homogeneous env.

	cine	monopoly	chess
PEDRO	2.26	3.04	0.30
GMOD	1.85	3.47	0.48
DMBO	5.66	9.95	3.37
DTC	11.33	19.79	62.60
GTCOD	24.00	14.93	42.42

(f) 80+1 heterogeneous env.

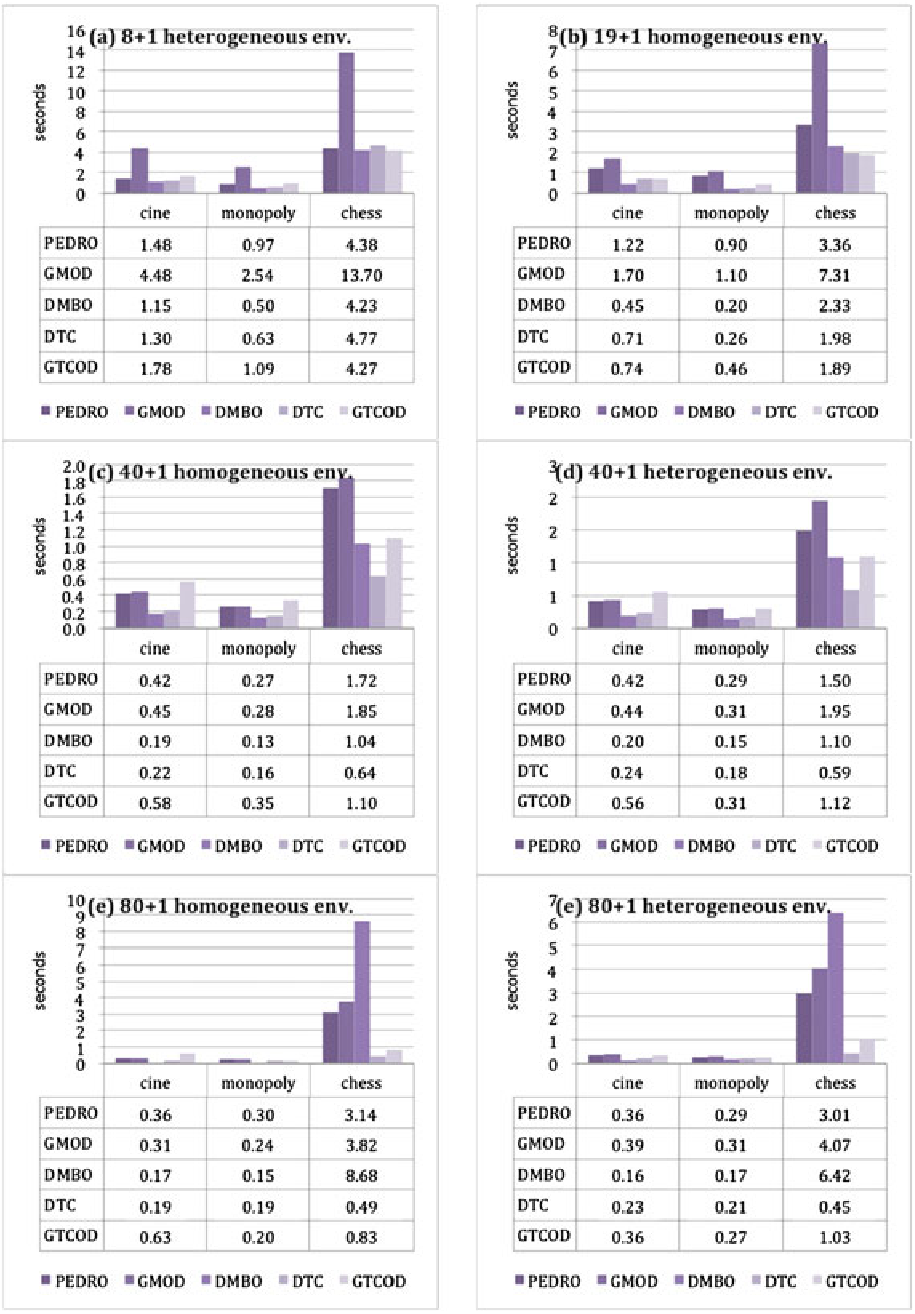
	cine	monopoly	chess
PEDRO	2.02	3.84	1.03
GMOD	2.84	3.81	0.64
DMBO	5.33	12.93	5.26
DTC	12.13	23.90	53.05
GTCOD	23.88	18.19	135.48

Figure 18. Waiting times.

cases. This means that at the end of the execution process, some *remoteExecutor* nodes have finished whereas others still have much work to do.

The second reason that these two algorithms are worse than the others is related to the problem described in Section 3. There is no way that a *remoteExecutor* node knows the mutants killed by other *remoteExecutor* nodes, and thus, it is possible that it executes a mutant that is already killed by another node. In this case, the mutant does not have to be executed, but it is. This problem makes the total number of execution increase with respect to other algorithms.

Figure 20 shows the average number of executions for each *remoteExecutor* node in each environment. It shows that in every case, the DTC and GTCOD algorithms ran more executions than the others, with GTCOD algorithm being worse than DTC algorithm.



(a) 8+1 heterogeneous env.

	cine	monopoly	chess
PEDRO	1.48	0.97	4.38
GMOD	4.48	2.54	13.70
DMBO	1.15	0.50	4.23
DTC	1.30	0.63	4.77
GTCOD	1.78	1.09	4.27

(b) 19+1 homogeneous env.

	cine	monopoly	chess
PEDRO	1.22	0.90	3.36
GMOD	1.70	1.10	7.31
DMBO	0.45	0.20	2.33
DTC	0.71	0.26	1.98
GTCOD	0.74	0.46	1.89

(c) 40+1 homogeneous env.

	cine	monopoly	chess
PEDRO	0.42	0.27	1.72
GMOD	0.45	0.28	1.85
DMBO	0.19	0.13	1.04
DTC	0.22	0.16	0.64
GTCOD	0.58	0.35	1.10

(d) 40+1 heterogeneous env.

	cine	monopoly	chess
PEDRO	0.42	0.29	1.50
GMOD	0.44	0.31	1.95
DMBO	0.20	0.15	1.10
DTC	0.24	0.18	0.59
GTCOD	0.56	0.31	1.12

(e) 80+1 homogeneous env.

	cine	monopoly	chess
PEDRO	0.36	0.30	3.14
GMOD	0.31	0.24	3.82
DMBO	0.17	0.15	8.68
DTC	0.19	0.19	0.49
GTCOD	0.63	0.20	0.83

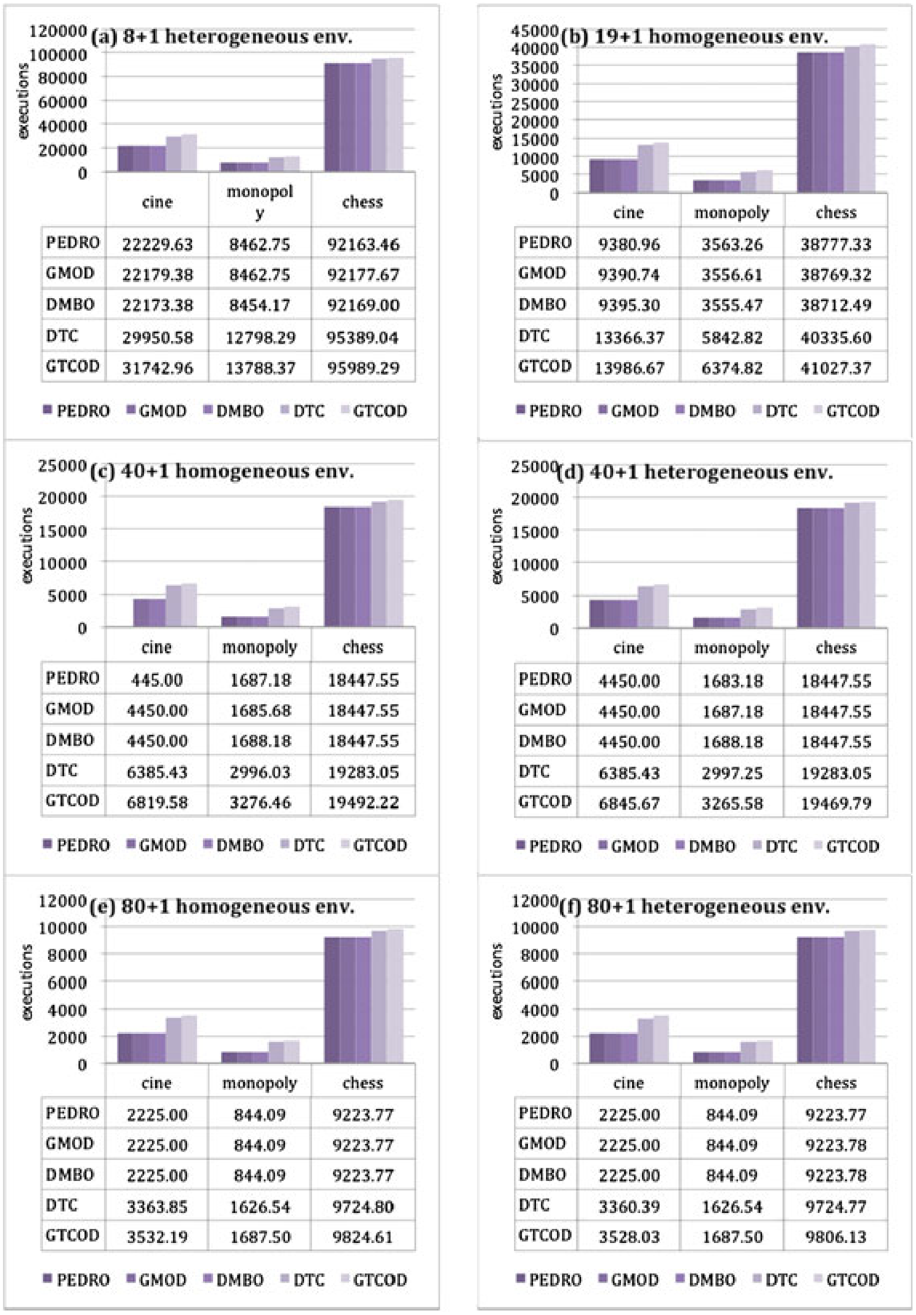
(e) 80+1 heterogeneous env.

	cine	monopoly	chess
PEDRO	0.36	0.29	3.01
GMOD	0.39	0.31	4.07
DMBO	0.16	0.17	6.42
DTC	0.23	0.21	0.45
GTCOD	0.36	0.27	1.03

Figure 19. Communication times.

Because DTC and GTCOD algorithms ran more executions and made a bad distribution of them, increasing the waiting times, their use is inadvisable.

Figure 20 also shows small differences in the executions preformed by PEDRO, DMBO and GMOD algorithms. These small differences are due to the timeouts. The Bacterio tool has a timeout parameter fixed at 3 s (Section 5.5). This is necessary for the some kinds of mutants whose mutation produces infinite loops. In these cases, when an ‘infinite-loop’ mutant is run, it runs until the execution time equals the timeout. At this point, the process is stopped, and the mutant is considered killed. However, the timeout has a problem that appears when a test case spends more time than the time established for the timeout with a normal mutant. In these cases, the mutant is considered



(a) 8+1 heterogeneous env.

	cine	monopol y	chess
PEDRO	22229.63	8462.75	92163.46
GMOD	22179.38	8462.75	92177.67
DMBO	22173.38	8454.17	92169.00
DTC	29950.58	12798.29	95389.04
GTCOD	31742.96	13788.37	95989.29

(b) 19+1 homogeneous env.

	cine	monopoly	chess
PEDRO	9380.96	3563.26	38777.33
GMOD	9390.74	3556.61	38769.32
DMBO	9395.30	3555.47	38712.49
DTC	13366.37	5842.82	40335.60
GTCOD	13986.67	6374.82	41027.37

(c) 40+1 homogeneous env.

	cine	monopoly	chess
PEDRO	445.00	1687.18	18447.55
GMOD	4450.00	1685.68	18447.55
DMBO	4450.00	1688.18	18447.55
DTC	6385.43	2996.03	19283.05
GTCOD	6819.58	3276.46	19492.22

(d) 40+1 heterogeneous env.

	cine	monopoly	chess
PEDRO	4450.00	1683.18	18447.55
GMOD	4450.00	1687.18	18447.55
DMBO	4450.00	1688.18	18447.55
DTC	6385.43	2997.25	19283.05
GTCOD	6845.67	3265.58	19469.79

(e) 80+1 homogeneous env.

	cine	monopoly	chess
PEDRO	2225.00	844.09	9223.77
GMOD	2225.00	844.09	9223.77
DMBO	2225.00	844.09	9223.77
DTC	3363.85	1626.54	9724.80
GTCOD	3532.19	1687.50	9824.61

(f) 80+1 heterogeneous env.

	cine	monopoly	chess
PEDRO	2225.00	844.09	9223.77
GMOD	2225.00	844.09	9223.78
DMBO	2225.00	844.09	9223.78
DTC	3360.39	1626.54	9724.77
GTCOD	3528.03	1687.50	9806.13

Figure 20. Number of executions.

killed. This problem is easily solved by increasing the timeout, but this increases the time for the total process.

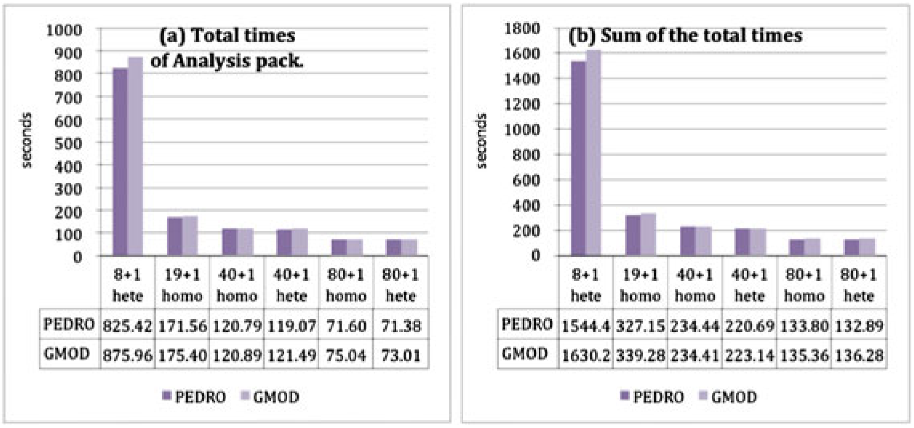
The small differences between the PEDRO, DMBO and GMOD algorithms in the executions must be due to the timeout. Some test cases can spend an amount of time that is very similar to the timeout with some mutants. At times, this gives small perturbations (produced by the operating system scheduler and other software running in the computers) enough influence to increase the execution time and to kill a mutant by timeout, which means that the remaining test cases do not run with this mutant.

The charts show that there are differences between the algorithms, with the best to worst being PEDRO and GMOD, DMBO, DTC and GTCOD algorithms. However, to increase the validity of the

Table XIV. Results of the Friedman test for all the algorithms.

Ranks		Friedman test	
Mean rank		*N*	18
PEDRO total time	1.44	Chi-square	61.689
GMOD total time	1.56	Degree of freedom	4
DMBO total time	3.22	Asymptotic significance	0.000
DTC total time	4.11	Exact significance	0.000
GTCOD total time	4.67	Point probability	0.000

PEDRO, Parallel Execution with Dynamic Ranking and Ordering; GMOD, Give Mutants On Demand; DMBO, Distribute Mutants Between Operators; DTC, Distribute Test Cases; GTCOD, Give Test Cases On Demand.



	8+1 hete	19+1 homo	40+1 homo	40+1 hete	80+1 homo	80+1 hete
PEDRO	825.42	171.56	120.79	119.07	71.60	71.38
GMOD	875.96	175.40	120.89	121.49	75.04	73.01

	8+1 hete	19+1 homo	40+1 homo	40+1 hete	80+1 homo	80+1 hete
PEDRO	1544.4	327.15	234.44	220.69	133.80	132.89
GMOD	1630.2	339.28	234.41	223.14	135.36	136.28

Figure 21. Total times of the Parallel Execution with Dynamic Ranking and Ordering (PEDRO) and Give Mutants On Demand (GMOD) algorithms.

conclusions, a statistical analysis was performed to compare all the algorithms and evaluate whether there are significant differences in the total time obtained with each. The selected statistical test used to compare the algorithms was the Friedman test (Section 5.6.2). The established hypotheses were H_0*: The distributions are the same across repeated measures with different treatments* and H_1*: The distribution across repeated measures is different*. Translated into this problem, the hypotheses are as follows:

H_0: The execution process time is similar with all algorithms; thus, any algorithm can be used.
H_1: The execution process time is different with each algorithm; thus, the best algorithm should be used.

The results obtained with the SPSS statistics software (IBM Corporation, Armonk, NY, USA) after applying the Friedman test are shown in Table XIV.

As Table XIV shows, the significance levels are 0. Therefore, the null hypothesis can be rejected. There are significant differences in the times obtained with each algorithm; thus, the best algorithms should be used.

6.2. comparison of the PEDRO and GMOD algorithms

The results of the previous section determined that the best algorithms were PEDRO and GMOD, but they were very similar. Thus, this section compares the total time obtained with these two algorithms, extended with a fourth application, the Analysis package (section 5.3), which is larger than the other applications.

Figure 21 shows the total times obtained with the fourth application, analysis package (Figure 21a) and the sum of the total times of all the applications in each environment (Figure 21b).

As Figure 21 shows, the PEDRO algorithm obtained better results than the GMOD algorithms in all cases with the analysis package. Also, Figure 21b shows that in almost all the cases, the PEDRO algorithm obtained better results than the GMOD algorithm. However, the differences between these two algorithms are lower when the number of nodes increases.

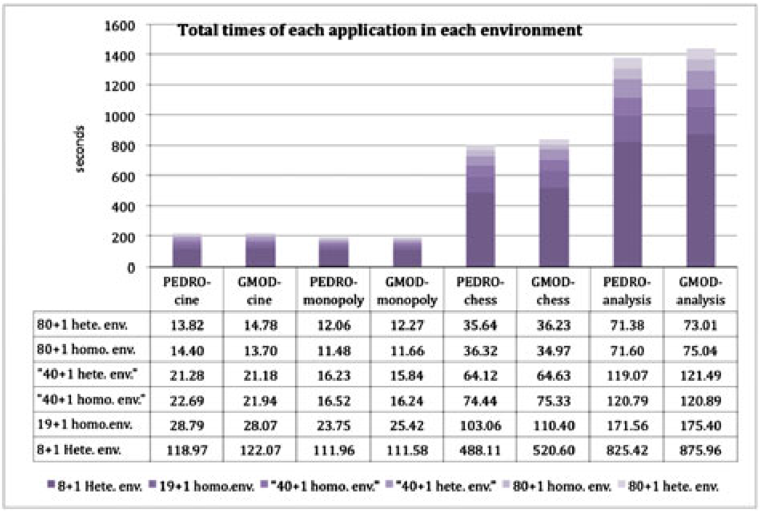
Figure 22 shows the times obtained by the two algorithms with each application and each environment. As in Figure 21, Figure 22 shows that the PEDRO algorithm obtained better results than the GMOD algorithm. Additionally, there is an increase in the differences when the application is larger. For example, with the smallest application (Cinema), the differences are between 0 and 5 s, but with the largest application (analysis math), the differences are between 0 and 56 s.

The results obtained and the nature of the PEDRO algorithm (it is automatically adapted to each environment taking into account the number of remote executor nodes or used parallel processors) mean that there is a direct relation between the differences in the PEDRO and GMOD algorithms and the expression '*Number of execution/Number of nodes*'. When this relation is bigger, the differences in the results for the PEDRO and GMOD algorithms are bigger, with PEDRO being the best algorithm. When the relation is lower, the differences in the results are lower, and the two algorithms become similar.

An example of this is the result obtained with Cinema in the $80 + 1$ homogeneous environment and the analysis pack in the $8+1$ environment. Table XV shows the number of executions necessary with each application obtained from multiplying the number of mutants in each application by the number of test cases.

In the first case, the relation '*Number of executions/Number of nodes*' equals $28\,560/80 = 357$, and the differences between the total time obtained by the algorithms is $14.40 - 13.70 = 0.7$ s, which means that the two algorithms are similar. However, in the second case, the relation '*Number of executions/Number of nodes*' equals $1\,221\,472/8 = 152\,684$, which is bigger than 357. In this case, the differences in the total time obtained are $825.42 - 875.96 = -50.54$ s, which means that the PEDRO algorithm is better than the GMOD algorithm.

After analysing the data collected from the experiments, the results show that PEDRO is the best algorithm, and in the cases when the relation '*Number of executions/Number of nodes*' is small, it is



	PEDRO-cine	GMOD-cine	PEDRO-monopoly	GMOD-monopoly	PEDRO-chess	GMOD-chess	PEDRO-analysis	GMOD-analysis
80+1 hete. env.	13.82	14.78	12.06	12.27	35.64	36.23	71.38	73.01
80+1 homo. env.	14.40	13.70	11.48	11.66	36.32	34.97	71.60	75.04
"40+1 hete. env."	21.28	21.18	16.23	15.84	64.12	64.63	119.07	121.49
"40+1 homo. env."	22.69	21.94	16.52	16.24	74.44	75.33	120.79	120.89
19+1 homo.env.	28.79	28.07	23.75	25.42	103.06	110.40	171.56	175.40
8+1 Hete. env.	118.97	122.07	111.96	111.58	488.11	520.60	825.42	875.96

Figure 22. Total times for each application in each environment for the Parallel Execution with Dynamic Ranking and Ordering (PEDRO) and Give Mutants On Demand (GMOD) algorithms.

Table XV. Number of executions with each application.

Cinema	Monopoly	Chess	Analysis pack
285 600	135 000	785 969	1 221 472

Table XVI. Wilcoxon signed-rank test comparing the GMOD–PEDRO total times paired values.

Ranks				Wilcoxon signed-rank tests	
	N	Mean rank	Sum of ranks	Z	−2.741
Negative ranks	4	3.75	15.00	Asymptotic significance (two-tailed)	0.006
Positive ranks	12	10.08	121.00	Exact significance (two-tailed)	0.004
Ties	0			Exact significance (one-tailed)	0.002
Total	16			Point probability	0.000

similar to the GMOD algorithm. However, to validate this conclusion, a statistical test was applied to compare the total times obtained with the two algorithms, PEDRO and GMOD, and to statistically determine whether there are significant differences.

The applied test was the Wilcoxon signed-rank test, designed to compare two related samples. This test has a null hypothesis that the two samples are from the same population, and the alternative hypothesis is that the two related samples are from different populations. In the study of this section, the related values are the total times obtained in the same environment with the same application but with different algorithms. The hypotheses are as follows:

H_0: The total times obtained with the two algorithms are similar.
H_1: The total times obtained with the two algorithms are different; thus, the PEDRO algorithm obtains better results.

The results from applying the Wilcoxon signed-rank test to the related samples of the GMOD–PEDRO values are shown in Table XVI:

As Table XVI shows, the significance levels are lower than 0.05. Thus, the null hypothesis can be rejected. There are significant differences between the total times obtained by the algorithms. Therefore, the PEDRO algorithm is the best algorithm to distribute execution work between nodes.

6.3. *Related studies*

After comparing the distribution algorithms, two other studies were carried out with the collected data: (i) study of the behaviour of the parallel execution technique when increasing the number of parallel nodes and (ii) study of the behaviour of the technique in homogeneous and heterogeneous environments.

6.3.1. Study of increasing the parallel nodes. With the parallel execution technique, obviously, when the number of parallel nodes increases, the results are better. If the same number of mutants and test cases are executed in one computer, it will take more time than if the same amount of work is executed in two computers. But efficiency does not increase linearly. When the number of computers grows significantly, a point is reached from which the efficiency does not increase but rather decreases.

Figure 23 shows two graphs that relate the total times obtained with the number of parallel nodes. The first shows the times obtained with each algorithm (these times are the sum of the times of all the applications), and the second shows the time obtained with each application using the PEDRO algorithm.

Figure 23a shows that despite the differences in the achieved times, all the algorithms have a similar behaviour, all describing an asymptotic curve. When the number of nodes is small, a small increase is enough to increase the efficiency very much. When there are many nodes, increasing their number has almost no influence on improving the efficiency. For example, the DTC algorithm spent approximately 1600 s with eight computers, which is more than 2500 s less than that when using a single computer. However, with the same algorithm, there are almost no differences in the times obtained with 40 or 80 computers.

Figure 23b shows that the times obtained with each application also describe the same behaviour, an asymptotic curve. This graphs shows that with the higher applications (understand 'higher' to be the application that produces more mutants), the curve decreases more slowly; thus, the efficiency

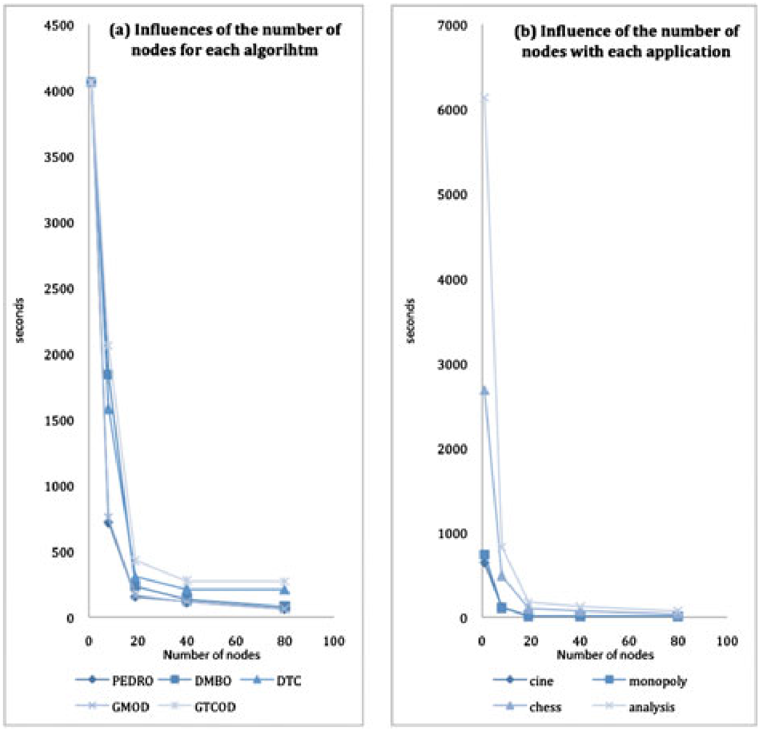


Figure 23. Total times versus number of parallel nodes.

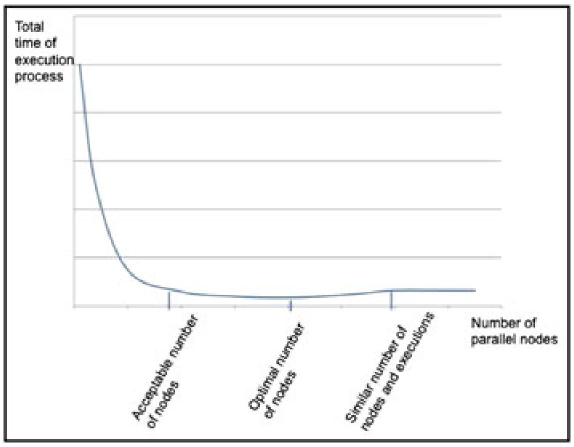


Figure 24. Theoretical model of increasing the nodes.

can be improved with more nodes. For example, with the Cinema application, after 20 nodes the efficiency almost does not increase, but with the Analysis package, the efficiency increases even with 80 nodes.

Figure 24 shows a theoretical model of how the total time varies with the number of parallel nodes. This can be split into three segments. In the first segment, the efficiency becomes better fast with the number of nodes. In the second segment, the efficiency becomes better slow with the number of nodes, and, depending on the nature of the application and the environment, the efficiency can become worse with a high number of nodes. In the third segment, the efficiency does not vary.

There are three important points in this theoretical model. The most important is the point that splits the first and second segments, 'Acceptable number of nodes'. This point determines an acceptable number of nodes that decreases the total execution time too much, and the number of parallel computers is low. This should be the selected number of nodes when the users have limited hardware resources.

The second important point is the point that splits the second and third segments, ‘Similar number of nodes and executions’. This point determines the number of parallel nodes from which the efficiency does not vary. Theoretically, at this point, the number of nodes is similar to the number of executions, but this assumption can be different depending on the distribution algorithm. For example, with the GMOD algorithm, this point is achieved when the number of nodes is equal to the number of mutants.

The third important point is the point that obtains the best results, ‘Optimal number of nodes’. This point should usually be the same point as the ‘Similar number of nodes and executions’, but if because of the nature of the application and the distribution algorithm the efficiency becomes worse with a high number of nodes, the ‘Optimal number of nodes’ point can be situated inside the second segment.

Although the collected data is not sufficient to obtain general conclusions (this study is a consequence of the study to compare algorithms), some conclusions were obtained related to the applications and environments used.

For relatively small applications, between 0 and 300 000 executions (executions are calculated multiplying the number of mutants by the number of test cases), an environment with 20 parallel nodes is sufficient to obtain good efficiency. For applications that need between 300 000 and 1 000 000 executions, an environment with 80 nodes is sufficient to obtain good efficiency. And for applications that need more than 1 000 000 executions, the environment must have more than 80 parallel nodes to obtain good efficiency. Unfortunately, the collected data are not sufficient to determine how many nodes would be enough.

However, a general conclusion can be obtained. The parallel execution technique decreases the computation costs of mutation quickly and effectively. Because it is the first segment that reduces the computation cost quickly, only a small number of computers is required to do this. For example, an environment with eight parallel computers was used in the experiments, the 8 + 1 heterogeneous environment. In the case of the Analysis package, the total time reduced the execution time 86.55% (from 6131 to 825 s).

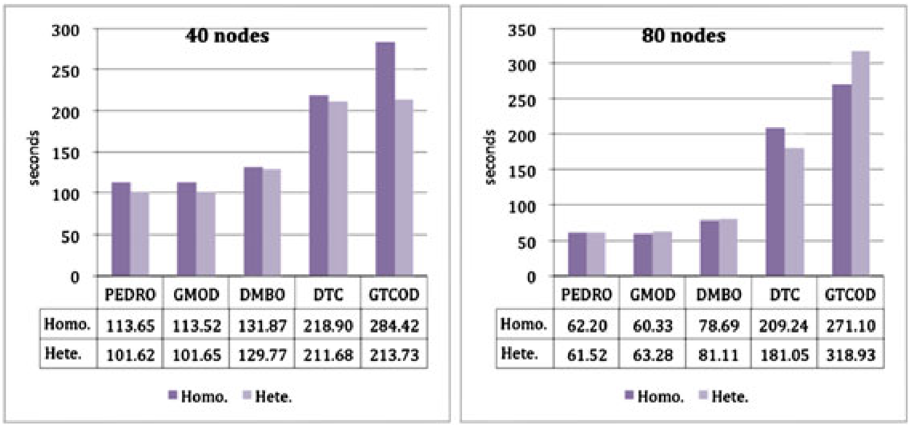
6.3.2. Homogeneous versus heterogeneous environments. The last study compares the effect of using a homogeneous environment or a heterogeneous environment. In this study, only the times obtained with the 40 + 1 homogeneous, 40 + 1 heterogeneous, 80 + 1 homogeneous and 80 + 1 heterogeneous environments were used because of their similarities in the network and in the computers. Figure 25 shows the sum of times obtained by each algorithm with all the applications. In Figure 25, there are two charts: one for the 40 + 1 environments and the other for the 80 + 1 environments. Each chart compares the total time obtained in the homogeneous configuration with the total time obtained in heterogeneous configuration.

As Figure 25 shows, with 40 nodes the homogeneous configuration takes more time than the heterogeneous configuration, but with 80 nodes, the opposite happens. However, except for the GTCOD algorithm, the differences are very small (between 12 and 1 s), and thus, this data were statistically analysed with the Wilcoxon signed-rank test, using as the two related values the total times obtained with the same application and with the same algorithm but a homogeneous or heterogeneous configuration. This test was applied three times: first with the data obtained with 40 nodes, then with the data obtained with 80 nodes and finally with all the data obtained. In the three cases, the following hypotheses were established:

H_0: The total times are similar in a homogenous environment and a heterogeneous environment.
H_1: The total times are different in a homogenous environment and a heterogeneous environment.

The statistical results of the three analyses are in Table XVII for 40 nodes, Table XVIII for 80 nodes and Table XIX for all the data.

Table XVII shows that the significance levels are closed to 0; thus, the null hypothesis can be rejected. There are significant differences in the total times obtained in a 40-node environment with



	PEDRO	GMOD	DMBO	DTC	GTCOD
Homo.	113.65	113.52	131.87	218.90	284.42
Hete.	101.62	101.65	129.77	211.68	213.73

	PEDRO	GMOD	DMBO	DTC	GTCOD
Homo.	62.20	60.33	78.69	209.24	271.10
Hete.	61.52	63.28	81.11	181.05	318.93

Figure 25. Comparison of the total times with a homogenous and a heterogeneous environment.

Table XVII. Wilcoxon test results for 40 nodes.

Ranks				Wilcoxon signed-rank tests	
	N	Mean rank	Sum of ranks	Z	−2.840
Negative ranks	14	7.86	110.00	Asymptotic significance (two-tailed)	0.005
Positive ranks	1	10.00	10.00	Exact significance (two-tailed)	0.003
Ties	0			Exact significance (one-tailed)	0.001
Total	15			Point probability	0.000

Table XVIII. Wilcoxon test results for 80 nodes.

Ranks				Wilcoxon signed-rank tests	
	N	Mean rank	Sum of ranks	Z	−0.966
Negative ranks	5	8.60	43.00	Asymptotic significance (two-tailed)	0.334
Positive ranks	10	7.70	77.00	Exact significance (two-tailed)	0.359
Ties	0			Exact significance (one-tailed)	0.180
Total	15			Point probability	0.014

Table XIX. Wilcoxon test results for 40 and 80 nodes.

Ranks				Wilcoxon signed-rank tests	
	N	Mean rank	Sum of ranks	Z	−1.388
Negative ranks	19	15.79	300.00	Asymptotic significance (two-tailed)	0.165
Positive ranks	11	15.00	165.00	Exact significance (two-tailed)	0.171
Ties	0			Exact significance (one-tailed)	0.085
Total	30			Point probability	0.003

a homogenous and heterogeneous configuration, with the heterogeneous configuration being better. However, for an environment with 80 nodes, Table XVIII shows that the significance levels are bigger than 0.05; thus, the null hypothesis cannot be rejected given that the total times are similar. The same happens when all the collected data are used. Table XIX shows that the significance levels using all the data are bigger than 0.05; therefore, on a general basis, the null hypothesis cannot be rejected.

The results of this study do not show empirical evidence to sustain that a homogenous or heterogeneous configuration has a significant influence on the total time. However, when the number of nodes increased, significant differences were observed between the two configurations. Thus, it would be recommendable to use a heterogeneous configuration only when the number of nodes is low.

6.4. *Threats to the validity*

This section explains the different threats to the validity of the experiments carried out and how they were mitigated.

6.4.1. Threats to construct validity. Construct validity is the degree to which the independent and dependent variables are accurately measured by the measurement instruments used in the experiments [50].

In the experiments here, all the independent variables are nominal; thus, they do not have to be measured. Related to the dependent variables, they are measured by the tool used to perform the experiments, Bacterio^{P}. As these variables are times measured in seconds, which are not deterministic because they depend on certain features of the hardware (such as processors, switches, network speed, etc.) and software (such as network protocols, operative system scheduler, other software inherent to the operating system, etc.), the times were measured three times and the mean was calculated to mitigate the threat. Also, the tool used can have bugs, which entails another threat to the construct validity.

6.4.2. Threats to the internal validity. Internal validity is the degree of confidence in a cause–effect relationship between a factor of interest and the observed results [50].

All the variables in the experiments were controlled to minimize the threats to the internal validity. To increase the confidence in the cause–effect relationship, the data were objectively analysed comparing it not only with charts and graphs but also with statistics to assure the cause–effect relationship.

6.4.3. Threats to the external validity. External validity is the degree to which the research results can be generalized to the population under study and other research settings [50]. The greater the external validity, the more the results of an empirical study can be generalized to actual software engineering practice.

The nature and the size of the samples influence the generalization of the results. This fact makes difficult to affirm that the obtained results and conclusions are conclusive. However, to mitigate these threads, applications with different sizes and natures were used in environments also with different sizes and natures, using a blocked subject-object study [50] covering all the possible combinations.

7. CONCLUSIONS

One of the most important problems with mutation is the cost of this testing technique. The parallel execution technique is a powerful technique to reduce the mutation computational cost, increasing the efficiency without less effectiveness. The work here describes an experimental comparison of the five distribution algorithms, one of them a novel contribution named the PEDRO algorithm, which is self-adaptive to the environment. The experiment demonstrates that the PEDRO algorithm is the best algorithm.

A study about how the number of parallel nodes affects efficiency was also carried out. From this study, a theoretical model was developed, and some recommendations were made on the basis of the experimental data. Finally, the effect that homogeneous and heterogeneous configurations have on efficiency was studied. The conclusion of this final study is that a heterogeneous configuration has better results than a homogeneous configuration only with small environments.

The parallel execution technique was studied with the current technologies. The conclusion is that this technique is very effective when it comes to reducing the computational cost of mutation testing because with some computers and appropriate tools, the computational costs are easily and quickly reduced.

As a future work, the study will be extended to validate the theoretical model described in Section 6.3.1, trying to test different applications with a larger set of environments with different numbers of nodes. It is also interesting to include more mutation operators, such as those described by Wong *et al.* [7].

ACKNOWLEDGEMENTS

This work was supported by the projects DIMITRI (TRA2009_0131) and PEGASO/MAGO (TIN2009-13718-C02-01) from MICINN. Pedro Reales has a doctoral grant (AP2009-3058) from the *Ministerio de Educación*.

The authors want also thank the QCyCAR group from the University of Castilla-La Mancha, especially professors Camelia Muñoz and Alfonso Niño, for their help with the use of the Hermes cluster, as well as Antonio Martín for his technical assistance.

REFERENCES

1. Offutt AJ. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology* 1992; **1**(1):15–20.
2. Polo M, Reales P. Mutation testing cost redution techniques: a survey. *IEEE Software* 2010; **27**(3):80–86.
3. Untch R, Offutt A, Harrold M. Mutation analysis using program schemata. In *International Symposium on Software Testing, and Analysis*: Cambridge, Massachusetts, June 28-30, 1993; 139–148.
4. King KN, Offutt AJ. A Fortran language system for mutation based software testing. *Software: Practice and Experience* 1991; **21**(7):685–718.
5. Barbosa EF, Maldonado JC, Marcelo A, Vincenzi R. Toward the determination of sufficient mutant operators for C. *Software Testing, Verification and Reliability* 2001; **11**(2):113–136.
6. Offutt AJ, Rothermel G, Untch RH, Zapf C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology* 1996; **5**(2):99–118.
7. Wong WE, Mathur AP. How strong is constrained mutation in fault deletion? *International Computer Symposium*, Taiwan, 1994; 515–520.
8. Wong WE, Maldonado JC, Delamaro ME, Mathur AP. Constrained mutation in C programs. *VIII Simpósio Brasileiro de Engenharia de Software (SBES 94)*, Curitiba, PR, Brazil, 1994; 439-452.
9. Polo M, Piattini M, García-Rodríguez I. Decreasing the cost of mutation testing with 2-order mutants. *Software Testing, Verification and Reliability* 2008; **19**(2):111–131.
10. Ma Y-S, Offutt J, Kwon YR. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability* 2005; **15**(2):97–133.
11. Offutt J. A practical system for mutation testing: help for the common programmer. *IEEE International Test Conference on TEST: The Next 25 Years*, Washington, DC, USA, 1994; 824–830.
12. Budd T. Mutation analysis of program test data. *Thesis in Yale University*, New Haven CT, 1980.
13. Commons A. Commons Math. Available from: http://commons.apache.org/math/ [last accessed 6 February 2011].
14. Mathur AP, Wong WE. An empirical comparison of data flow and mutation based test adequacy criteria. *Software Testing, Verification, and Reliability* 1994; **4**(1):9–31.
15. Ma YS, Kwon YR, Offutt J. Inter-class mutation operators for Java. *13th International Symposium on Software Reliability Engineering*, Annapolis, MD, 2002; 352–363.
16. Offutt AJ, Pargas RP, Fichter SV, Khambekar PK. Mutation testing of software using a MIMD computer. *International Conference on Parallel Processing*, 1992.
17. Choi B, Mathur A. High-performance mutation testing. *Journal of Systems and Software* 1993; **20**(2):135–152.
18. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* September 2011; **37**(5):649–678.
19. Mathur AP, Krauser EW. Modeling mutation on a vector processor. *10th International Conference on Software Engineering*, Singapore, 11-15 April, 1988; 154– 161.
20. Krauser EW, Mathur AP, Rego VJ. High performance software testing on SIMD machine. *IEEE Transactions on Software Engineering* 1991; **17**(5):403–423.
21. Rego V, Mathur AP. Concurrency enhancement through program unification: a performance analysis. *Parallel and Distributed Computing* 1990; **8**(3):201–217.
22. DeMillo RA, Guindi DS, King KN, McCracken WM, Offutt AJ. An extended overview of the Mothra software testing environment. *Second Workshop on Software Testing, Verification and Analysis*, Banff, Alberta, Canada, 1988; 142–151.
23. Weiss SN, Fleyshgakker VN. Improved serial algorithms for mutation analysis. In *International Symposium on Software Testing and Analysis (ISSTA)*. Cambridge: Massachusetts, USA, 1993; 149–158.
24. Fleyshgakker VN, Weiss SN. Efficient mutation analysis: a new approach. In *International Symposium on Software Testing and Analysis*: Seattle, Washington, United States, August 17-19, 1994; 185–195.
25. Ammann P, Offutt J. *Introduction to software testing*. Cambridge University Press: Cambridge, UK, 2008.
26. Díaz J, Reyes S, Niño A, Muñoz-Caro C. Derivation of self-scheduling algorithms for heterogeneous distributed computer systems: application to internet-based grids of computers. *Future Generation Computer Systems* 2009; **25**(6):617–626.
27. Polychronopoulos CD, Kuck DJ. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers* December 1987; **36**(12):1425–1439.
28. Hummel S F, Schonberg E, Flynn L E. Factoring: a method for scheduling parallel loops. *Magazine Communications of the ACM* August 1992; **35**(8):90–101.

29. Hummel SF, Schmidt J, Uma RN, Wein J. Load-sharing in heterogeneous systems via weighted factoring. *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1996; 318–328. SPAA.
30. Banicescu I, Liu Z. Adaptive factoring: a dynamic scheduling method tuned to the rate of weight changes. *Proceedings of the High Performance Computing Symposium*, Washington, D.C., April 2000; 122–129. HPC 2000.
31. Tzen TH, Ni LM. Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems* January 1993; **4**(1):87–98.
32. Casavant TL, Kuhl JG. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering* February 1988; **14**(2):141–154.
33. Reales P, Polo M, Offutt J. Mutation at system and functional levels. *Third International Conference on Software Testing, Verification, and Validation Workshops*, Paris, France, April 2010; 110-119.
34. Kapfhammer G. M. Automatically and transparently distributing the execution of regression test suites. *18th International Conference on Testing Computer Software*, Washington, D.C., 31 May, 2001.
35. DeMillo R, Lipton RJ, Sayward FG. Hints on test data selection: help for the practicing programmer. *IEEE Computer* 1978; **11**(4):34–41.
36. Bombieri N, Fummi F, Pravadelli G, Hampton M, Letombe F. Functional qualification of TLM verification. In *Design, Automation and Test in Europe, DATE'09,* Nice, France April 20-24, 2009; 190–195.
37. Offutt AJ, Lee SD. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering* 1994; **20**(5):337–344.
38. JODE: Java Optimize and Decompile Environment. Availablefrom:http://jode.sourceforge.net/ [last accessed 21 May 2010].
39. JUnit, testing resources for extreme programming, 2003. Availablefrom:http://www.junit.org [last accessed 5 January 2003].
40. Medina R, Pratmarty P. UISpec4J: Java/Swing GUI testing made simple!. Available from: http://www.uispec4j.org/ [last accessed 21 September 2009].
41. ORACLE. Remote method invocation home. Availablefrom:http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html [last accessed May 2011].
42. Ghan AAA, Wei KT, Muketha GM, Wen WP. Complexity metrics for measuring the understandability and maintainability of business process models using goal–question–metric (GQM). *International Journal of Computer Science and Network Security* 2008; **8**(5):219–225.
43. Alarcos Research Group. Available from: http://alarcos.inf-cr.uclm.es [last accessed December 2011].
44. QCyCAR – Grupo de Quìmica Computacional y Computaciòn de Alto Rendimiento. Available from: http://qcycar-uclm.esi.uclm.es/index.php?lang=english [last accessed December 2011].
45. Walton L. Eclipse metrics plugin. Available from: http://eclipse-metrics.sourceforge.net/ [last accessed 10 February 2011].
46. Hoffmann MR. Eclemma code coverage analysis for Eclipse, 2007. Available from: http://www.eclemma.org [last accessed February 2011].
47. Eclipse framework. Available from: http://www.eclipse.org/ [last accessed 10 February 2011].
48. Conover WJ. *Practical Nonparametric Statistics*, 3rd Edition. John Wiley & Sons: UK, 1999.
49. Wilcoxon F. Individual comparisons by ranking methods. *Biometrics Bulletin* December 1945; **1**(6):80–83.
50. Wohlin C, Runeson PM, Höst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation In Software Engineering: An Introduction*, The Kluwer International Series In Software Engineering, Vol. 6. Kluwer Academic Publishers: Norwell, Massachusetts, USA, 2000.